2020 JC1 H2 Computing 9569

01. Data Types and Basic Operations

Variables

Variable is a name declared in a program to represent something. It can be one of the many types of entities, such as:

- a value
- a set of data
- a file
- another program

For a start, we will deal with values. Once a variable is created, the value associated with it can retrieved or modified.

Naming Variables

When writing a program, we often need to assign names to different variables. As there can be many different variables within the same program, it is important to give them names that are helpful, so that the program is more readable. As we gain more experience, we will have a sense of what kinds of names are helpful or not. Below is a guideline of naming variables.

1. Every name must begin with a letter or an underscore (_).

    - A numeral is not allowed as the first character.
    - Multiple-word names can be linked together using the underscore, e.g. my_name, the_best_is_yet_to_be.
    - A name that actually starts with an underscore is usually used to denote a variable with special characteristics.
      It is not recommended to start variable names with an underscore for now.

2. After the first letter, the name may contain any combination of letters, numbers and underscores.

    - The name can be of any length.
    - The name cannot be a keyword.
    - The name cannot contain any delimiters (e.g. parentheses), punctuation, or operators (e.g. +, -, *,).
    - The name is case sensitive, i.e. acjc_1977 is different from Acjc_1977 or ACJC_1977.

Keywords

Keywords are words used by Python to indicate commands to the interpreter. As such, they are reserved and cannot be used as variable names. Here is an (incomplete) list of keywords.

```
and as assert break class continue def del elif else except exec
False finally for from global if import in is lambda None not or
```

```
pass print raise return True try while with yield
```

Objects and Types

In Python, every 'thing' is considered to be an object that has:

- an identity
- some attributes
- some (possibly zero) names

When an object is created by Python, it receives an ID number. The **id()** function can be used to discover its ID number. This ID number is system-dependent, which means it will differ from machine to machine or even from session to session on the same machine. It actually indicates the location where the object is stored in the computer's memory. This number is difficult for humans to remember, which is precisely why we give names to variables.

To execute a Python program code in Jupyter Notebook, we need to press *Shift+Enter* on the relevant cell.

When we want to restart the system or the notebook stops working (indicated by an asterisk (*) next to a program code cell), we can click on *Restart & Clear Output* under the *Kernel* tab.

To put a non-executable comment in Python, we use the hash character (#).

In [ ]:
```
x = 1
id(x)
```

The **print()** function is used to display the given object to the standard output device, i.e. computer screen.

In [ ]:
```
x = 2
y = 3

# To print the contents of the variables
print(x)
print(y)

# To print the memory addresses of the variables
print(id(x))
print(id(y))
```

The following is the way to assigning multiple values to multiple variables in one line.

In [ ]:
```
x, y = 5, 7

print(x)
print(y)
```

An important attribute of an object is its **type**, which dictates the data stored in it.

These are the more common types that we will encounter for now.

Type	Description	Examples

Type	Description	Examples
int (integer)	This corresponds to our mathematical definition of integer. They can be positive or negative.	1, 23, 142857, -1886
float (floating-point number)	This roughly corresponds to real numbers.	2.71828, -3.5
bool (Boolean value)	True or False	True, False
str (string)	A collection of characters in a sequence, delimited by single quotes (') or double quotes (")	'This is a string!', 'i am groot', 'x'
list	A mutable collection of objects in a sequence, delimited by square brackets ([ ]). The objects do not need to be all of the same type.	[4, 3.14, 'abc'], ['a', 'b', ['c', 'd'], 'e']
tuple	An immutable collection of objects in a sequence, delimited by square brackets (( )). The objects do not need to be all of the same type.	(4, 3.14, 'abc'), ('a', 'b', ('c', 'd'), 'e')
set	A collection of unique elements, delimited by curly brackets ({ }). The objects do not need to be all of the same type, and their order do not matter.	{4, 3.14, 'abc'}
dict (dictionary)	A set of element pairs. The first element in each pair is the **key** and the second is the **value**. The key can be used to look up the value.	{'Jones':3471124, 'Larson':3472289, 'Smith':3471288}

Note that in other programming languages, such as Java and C++, there is a **char** data type. It is essentially a one-character string.

We can check they type of an object using the **type()** function.

In [ ]:
```
type(5)
```

In [ ]:
```
type(3.1)
```

In [ ]:
```
type(3.0)
```

In [ ]:
```
type(True)
```

In [ ]:
```
type('abc')
```

In [ ]:
```
type(x)
```

In [ ]:
```
y = 'Hello'
type(y)
```

Typecasting

In some cases, it is possible to change the type of an object. Known as **typecasting**, Python will make a new object of the specified type that is most similar to the given object as far as possible.

Take a look at a few examples below. What do you notice when a float is cast into an integer?

In [ ]:
```
float(3)
```

In [ ]:
```
int(2.1)
```

In [ ]:
```
int(2.9)
```

In [ ]:
```
int(-0.9)
```

Does executing *int(var)* change the type of variable *var*?

In [ ]:
```
x = 1.6
print(int(x))
print(type(x))
```

In [ ]:
```
y = int(x)
print(type(y))
```

In [ ]:
```
print(str(1965))

s = '2000'
print(int(s))
print(float(s))
print(type(s))
```

Typecasting involving Boolean variables is more complicated and will be covered in the next topic.

Operators

We carry out **operations** on objects to get a result. For instance, applying the operation of addition to the integers 2 and 3 gives us the integer 5.

Many of the operators for integers and floats are precisely what we would expect from Mathematics.

In [ ]:
```
2+3
```

In [ ]:
```
2-3
```

In [ ]:
```
2*3
```

In [ ]:
```
2/3
```

Note that the result of dividing an integer by another is always a float, even if the division does not have a remainder.

In [ ]:
```
10/5
```

The operator for exponentiation is given by double asterisks (**).

In [ ]:
```
2 ** 3
```

Python also performs modular arithmetic, that is, division with a quotient and remainder.

Double slashes (//) gives us the quotient, while the percentage character (%) gives us the remainder.

In [ ]:
```
x = 36
y = 7
```

```
q = x // y
r = x % y
s = x / y

print(q)
print(r)
print(s)

print(type(q))
print(type(r))
print(type(s))
```

Parentheses and BODMAS (Bracket, Of, Division, Multiplication, Addition and Subtraction) rule work as they would in Mathematics.

In [ ]:
```
3 + 2 * 5
```

In [ ]:
```
(3 + 2) * 5
```

In [ ]:
```
2 * 3 ** 2
```

Note that when we mix integers and floats, the result is always a float.

In [ ]:
```
a = 12 + 3.0
b = 12 * 3.0

print(a)
print(type(a))

print(b)
print(type(b))
```

In [ ]:
```
x = 36
y = 3.5

q = x // y
r = x % y
s = x / y

print(type(q))
print(type(r))
print(type(s))
```

Reassigning Variables

In the course of writing a program, we may need to change the value of a variable. The most straightforward way is to assign it a new value.

In [ ]:
```
x = 3
print(x)

x = 5
print(x)
```

The new value may depend on the previous one.

In [ ]:
```
x = 3
print(x)

x = x+1
```

```
print(x)

x = x+1
print(x)
```

We can also swap the values of two (or more) variables.

In [ ]:
```
a, b = 3, 5
print(a)
print(b)

a, b = b, a
print(a)
print(b)
```

Why is the following program code not working as intended?

In [ ]:
```
a, b = 3, 5
print(a)
print(b)

a = b
b = a
print(a)
print(b)
```

2020 JC1 H2 Computing 9569

02. Boolean

Boolean is a special kind of variable that only takes on one of two possible values: True or False.

In [ ]:
```
x = True
print(type(x))
print(x)

y = False
print(type(y))
print(y)
```

Comparison Operators

We have the following symbols that are used to compare two numbers (integers or floats).

Symbol	Meaning
<	less than
>	greater than
==	equal to
!=	not equal to
<=	less than or equal to
>=	greater than or equal to

In [ ]:
```
x = 2==2
print(type(x))
print(x)
```

In [ ]:
```
a = 2
x = (a == 2)
print(x)
```

In [ ]:
```
y = (a > 1)
print(y)
```

In [ ]:
```
z = (a < 1)
print(z)
```

In [ ]:
```
p = (a >= 2)
print(p)
```

In [ ]:
```
q = (a != 2)
print(q)
```

In [ ]:
```
r = (a != 3)
print(r)
```

Note that two strings can also be compared with each other according to **lexicographic**

order.

The convention used in Python is that the space (' ') comes first, followed by the digits '1' to '9', followed by the capital letters 'A' to 'Z', and finally the lowercase letters 'a' to 'z'.

In [ ]:
```
' ' < '1'
```

In [ ]:
```
'1' < 'A'
```

In [ ]:
```
'Z' < 'a'
```

In [ ]:
```
'a' < 'b'
```

In [ ]:
```
'an' < 'at'
```

In [ ]:
```
'ant' < 'antman'
```

In [ ]:
```
'ant' < 'Antman'
```

Boolean Operators

Sometimes we need to work with two Boolean variables to make a third.

The usual operators for doing so are the **not**, **and** and **or** operators.

1. not

x	not x
True	False
False	True

In essence, *not x* is the opposite of *x*.

2. and

x	y	x and y
True	True	True
True	False	False
False	True	False
False	False	False

In summary, *x and y* is True only when both *x* is True and *y* is True.

3. or

x	y	x or y
True	True	True
True	False	True

x	y	x or y
False	True	True
False	False	False

In summary, *x or y* is True when *x* is True, or *y* is True, or both.

Note that in Computing (and Mathematics) the word 'or' always includes the case when both statements are True. This is sometimes called the **inclusive or**.

There is an old joke where a waiter asks a programmer, "Would you like coffee or tea?"
The programmer says, "Yes."

Try to work out the following codes.

In [ ]:
```
x = 2
y = 3
a = (x < 1) or (y < 4)
print(a)
```

In [ ]:
```
b = (x == 2) and (y > 3)
print(b)
```

In [ ]:
```
c = (x + y >= 4) and (y - x <= 0)
print(c)
```

Exercise

Change *x* to different integer values to check if you have done your program codes correctly.

In [ ]:
```
x = 2021
```

Define a variable *a* so that it is True when *x* is even and False when x is odd.

In [ ]:
```
a = (x % 2 == 0)
print(a)
```

Define a variable *b* so that it is True when *x* is odd and False when *x* is even.

In [ ]:
```
b = (x % 2 == 1)
print(b)
# OR if we want to reuse the variable a: b = not a
```

Define a variable *c* so that it is True when *x* is an odd multiple of 3 and False otherwise.

What kinds of values should we try for *x*?

In [ ]:
```
c = (x % 2 == 1) and (x % 3 == 0)
print(c)
```

Define a variable *d* so that it is True when *x* is a multiple of 4, but not of 100.

For example, when x == 96, *d* is True but when x == 100, *d* is False.

What kinds of values should you try for *x*?

In [ ]:
```
d = (x % 4 == 0) and (x % 100 != 0)
print(d)
```

The Gregorian Calendar, which we are currently using, has leap years when the year is a multiple of 4, except when the year is a multiple of 100. Interestingly, it also has leap years when the year is a multiple of 400.

For example, the years 1892, 1904, 1992 and 2004 were leap years (since they are multiples of 4, but not of 100). The years 1700, 1800 and 1900 were not (since they are multiples of 100, but not of 400), and neither will 2100. However, 1600 and 2000 were leap years (since they are multiples of 400).

Write a program code to determine whether *y* is a leap year. The variable *isleap* should be True when *y* is a leap year and False otherwise.

In [ ]:
```
y = 2000
isleap = (y % 400 == 0) or ((y % 4 == 0) and (y % 100 != 0))
print(isleap)
```

Typecasting

Typecasting into Boolean

The following table summarises how other data types are typecast into Boolean variables.

Data type	False	True
int or float	0 or 0.0	all other values
str	empty string ''	all other strings (including 'False'!)
list	empty list []	all other lists
set	empty set {}	all other sets

Typecasting from Boolean

The following table summarises how other data types are typecast from Boolean variables.

Data type	False	True
int	0	1
float	0.0	1.0
str	'False'	'True'
list or set	error	error

Use the space below to experiment with typecasting.

In [ ]:
```
# Type your code here
```

For Boolean operators, Python performs some typecasting automatically and this allows us to shorten some codes.

For instance, in the example above where we had to define the variable *a* to determine whether or not *x* was even, we could have done the following.

In [ ]:
```
x = 2019
a = not(x % 2)
print(a)
```

While this makes the code more concise, it also makes it much harder for humans to read. Therefore, it is still advisable not to typecast Boolean variables, but instead use the operators defined above.

Conditional Statements

1. If statement

The basic structure of the **if statement** is as follows.

In [ ]:
```
if <boolean_expression>:
    #code1
#code2
```

It does the following:

1. Evaluate to determine whether it is True or False.

2. If is True,
    - Execute the **indented** code under the if (i.e. in the position of #code1).
    - Once that is done, continue with any code after the indented code (i.e. in the position of #code2).

3. If is False,
    - Ignore any indented code under the if (i.e. in the position of #code1).
    - Continue running any code after the indented code (i.e. in the position of #code2).

In [ ]:
```
x = -3

if x < 0:
    print("The value of x has been changed to zero.")

print("This is the end of the program.")
```

2. If-else statement

The basic structure of the **if-else statement** is as follows.

In [ ]:
```
if <boolean_expression>:
    #code1
else:
    #code2
#code3
```

It does the following:

1. Evaluate to determine whether it is True or False.

2. If is True,
    - Execute the indented code under the if (i.e. in the position of #code1).
    - Once that is done, continue with any code after all the indented code (i.e. in the position of #code3).

3. If is False,
    - Execute the indented code under the else (i.e., in the position of #code2).

- Once that is done, continue with any code after all the indented code (i.e. in the position of #code3).

In [ ]:
```
x = 3
if x % 2 == 0:
    print("x is even.")
else:
    print("x is odd.")

print("That's all, folks!")
```

Complete the code below so that it prints "The first integer is bigger." or "The second integer is bigger." as appropriate.

Assume that the two integers are never equal.

In [ ]:
```
first_int = 20
second_int = 20

if first_int > second_int:
    print("The first integer is bigger.")
else:
    print("The second integer is bigger.")
```

3. Nested statement

If and if-else statements can be nested inside one another.

In [ ]:
```
x = 5

if x % 3 == 0:
    print("x is a multiple of 3.")
else:
    if x % 3 == 1:
        print("x has a remainder of 1 when divided by 3.")
    else:
        print("x has a remainder of 2 when divided by 3.")
```

Complete the code below so that it prints "It is a leap year." or "It is not a leap year." as appropriate.

In [ ]:
```
year = 2004

if year % 400 == 0:
    print("It is a leap year.")
else:
    if year % 100 == 0:
        print("It is not a leap year.")
    else:
        if year % 4 == 0:
            print("It is a leap year.")
        else:
            print("It is not a leap year.")
```

4. If-elif-else statement

The **elif statement** helps us to avoid too many layers of nesting, which make a program code hard to read.

The basic structure of the **if-elif-else statement** is as follows.

In [ ]:
```
if <boolean_expression1>:
    #code1
elif <boolean_expression2>:
    #code2
elif <boolean_expression3>:
    #code3

# There can be as many elif statements as needed.

else:
    #code_else
#code_after
```

It does the following:

1. Evaluate to determine whether it is True or False.

2. If is True,
    - Execute the indented code under the if (i.e. in the position of #code1).
    - Once that is done, continue with any code after the all indented code (i.e. in the position of #code_last).

3. If is False,
    - Evaluate to determine whether it is True or False.
    - If is True,
        - Execute the indented code under the elif (i.e. in the position of #code2).
        - Once that is done, continue with any code after the all indented code (i.e. in the position of #code_last).
    - If is False,
        - Evaluate to determine whether it is True or False.
        - If is True,
            A. Execute the indented code under the elif (i.e. in the position of #code3).
            B. Once that is done, continue with any code after the all indented code (i.e. in the position of #codelast).
        - If is False,
            A. etc.

In other words, the code goes down , , etc. in order, until it finds the first one which is evaluated to be True and subsequently executes the indented code under that particular elif. After that, it continues on to the unindented code.

The example below shows how the code from earlier can be rewritten using elif.

In [ ]:
```
x = 5

if x % 3 == 0:
    print("x is a multiple of 3.")
elif x % 3 == 1:
    print("x has a remainder of 1 when divided by 3.")
else:
    print("x has a remainder of 2 when divided by 3.")
```

Try to use elif to write a code that prints "It is a leap year." or "It is not a leap year." as appropriate.

In [ ]:

```
year = 2004

if year % 400 == 0:
    print("It is a leap year.")
elif year % 100 == 0:
    print("It is not a leap year.")
elif year % 4 == 0:
    print("It is a leap year.")
else:
    print("It is not a leap year.")
```

2020 JC1 H2 Computing 9569

03. String

String is a built-in data type in Python used to represent text.

The following are some examples of strings:

- "This is a string"
- 'This is also a string'
- "125"

Note that strings are always enclosed within a pair of inverted commas, either ' ' or " ".

Try to write a program code to print the following:

```
Mr Cliff says, "Hello, everyone! Welcome to Computing!"
```

In [ ]:
```
print('Mr Cliff says, "Hello, everyone! Welcome to Computing!"')
```

String Concatenation

String concatenation allows us to join two or more strings together.

In [ ]:
```
print("a" + "b")
```

In [ ]:
```
result = "one" + "one" + "one"
print(result)
```

In [ ]:
```
# What does this do?
test = "abc"
print(test * 3)
```

String Indexing and Slicing

String slicing allows us to extract specific portions of a string. It has the following format.

```
<name_of_string>[<start>:<stop>:<step>]
```

Only *start* is compulsory within the square brackets ([ ]).

In [1]:
```
s = "some random string"
print(s[0])
print(s[7])
```

```
s
n
```

In the program code above, the variable *s* is assigned the string "some random string". The number 0 enclosed within [ ] indicates the **index** (position) of the character we want to extract.

Note that Python (and most programming languages) uses **zero indexing**. In other words, the first item has index number 0. Hence, to extract the first character of a string, we use *s[0]*.

A space (' ') is also regarded as a character. Hence, *s[7]* extracts the 8th character in this string, which is 'n'.

In [2]:
```
print(s[0:4])
print(s[3:8])
```

```
some
e ran
```

The program code above shows the use of both *start* and *stop* indexes. The strings are sliced from the *start* index up to (but excluding) the *stop* index.

In [3]:
```
print(s[0:10:2])
print(s[0:10:3])
```

```
sm ad
seao
```

The program code above shows the use of *start*, *stop* and *step* indexes. The strings are sliced from the *start* index up to (but excluding) the *stop* index, advancing *step* indexes every time.

It is also possible to slice strings using negative values. Experiment with the following to see how it works.

- `s[-1]`
- `s[10:0:-1]`
- `s[-10:-2:1]`

In [4]:
```
print(s[-1])
print(s[10:0:-1])
print(s[-10:-2:1])
```

```
g
modnar emo
dom stri
```

Common String Functions and Methods

Function or Method	Description	Example	Output
len(s)	Returns the length of the string	s = 'abc' len(s)	3
s1 in s2	Checks if s1 is in s2	s1 = 'abc' s2 = 'abcdefg' s1 in s2	True
s1 == s2	Checks if s1 is equivalent to s2	s1 = 'abc' s2 = 'abc' s1 == s2	True
str(n)	Converts n into a string	str(14)	'14'
.index("")	Returns first index where given substring can be found in the str or raises ValueError if	"Computing".index("in")	True

	given substring is not found		
.isalnum()	Returns whether the str is made of alphanumeric characters only	"Computing".isalnum()	True
.isalpha()	Returns whether the str is made of alphabetical characters only	"Computing".isalpha()	True
.isdigit()	Returns whether the str is made of numerical digits only	"Computing".isdigit()	False
.isspace()	Returns whether the str is made of whitespace characters only	"Computing".isspace()	False
.isupper()	Returns whether the str is made of uppercase characters only	"Computing".isupper()	False
islower()	Returns whether the str is made of lowercase characters only	"Computing".islower()	False
.startswith("")	Returns whether the str starts with the given prefix	"Computing".startswith("g")	False
.endswith("")	Returns whether the str ends with the given suffix	"Computing".endswith("g")	True
.lower()	Returns the str converted to lowercase	"Computing".lower()	"computing"
.upper()	Returns the str converted to uppercase	"Computing".upper()	"COMPUTING"

You can use the space below to experiment with the different functions and methods.

In [ ]:

String Formatting

The **.format()** method gives us an alternative way of concatenating strings.

In [ ]:
```
str1 = "How {} you?".format("are")
print(str1)
```

In [ ]:
```
str2 = "1 + 1 = {}".format(1+1)
print(str2)
```

In [ ]:
```
str3 ="{} and {}".format("Apple", "Pen")
print(str3)
```

Manual numbering within { } is possible. The number indicates the order of placement of the strings.

In [ ]:
```
str4 = "1st: {1}, 2nd: {0}".format("one", "two")
print(str4)
```

In [ ]:
```
str5 = "{1}, {2}, {0}".format("One", "Two", "Three")
print(str5)
```

User Input

It is common for a program to ask for input from the user to perform specific task(s).

Note that any input by the user is stored as a string.

In [ ]:
```
name = input("What is your name? ")
print("Hello, " + name + "! Nice to meet you!")
```

Does the program code below work as what you would expect?

In [ ]:
```
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

answer = num1 + num2

print("The answer is " + str(answer))
```

Exercise

Write a program code that:

- asks the user for an input string
- display the string in the reverse order

In [ ]:
```
str1 = input("Enter your string here: ")
result = str1[::-1]
print(result)
```

Appendix

There are other operations and methods that we can perform on strings.

Refer to the official documentation for a more complete list.

https://docs.python.org/3.6/library/stdtypes.html#typesseq-common

2020 JC1 H2 Computing 9569
04. Flowchart and Decision Table

An **algorithm** is a sequence of steps to complete a particular task.

As a simple example, the following is one algorithm to prepare a slice of bread with jam for breakfast.

1. Take a slice of bread from the packet of bread.
2. Grab a bottle of jam from the fridge.
3. Open the bottle of jam.
4. Scoop some jam from the bottle using a knife.
5. Spread the jam on the bread using the knife.
6. Fold the bread in half.

Very often, there is more than one way to complete a task. Hence, one can formulate multiple algorithms to solve the same problem. However, some algorithms are more efficient than others. We shall discuss about efficiency in a later topic.

Flowchart

A **flowchart** is a graphical representation of a computer program in relation to the sequence of steps it is intended to perform.

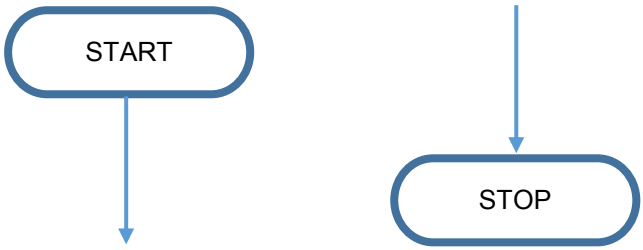
There are four standard symbols used in flowcharts.

1. Terminator

The terminator symbol is a rectangle with rounded corners.

It represents either:

- the beginning of the algorithm with the START command, or
- the end of the algorithm with the STOP command.



2. Data

The data symbol is a parallelogram.

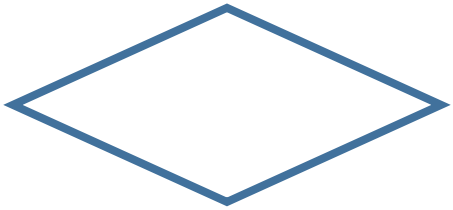
It represents a step of either:

- receiving input data from outside the algorithm using the INPUT command,
- or producing output from within the algorithm using the OUTPUT command.

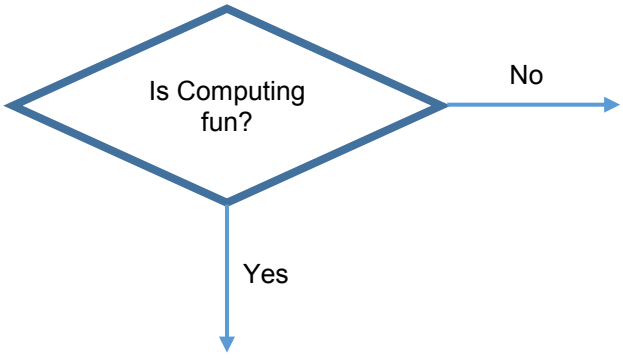
An example of each type of command is shown below.

3. Decision

The decision symbol is a diamond.



It represents a step involving a question. The outgoing arrows represent the possible outcomes to the question and are usually labelled “Yes” and “No”. There may be two or three outgoing arrows depending on the number of possible outcomes. Only one of these outgoing arrows should be followed at any one time when performing the algorithm.

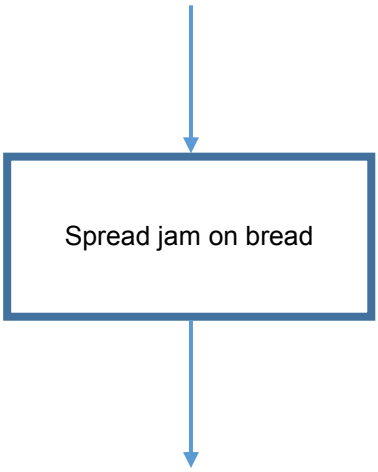


4. Process

The process symbol is a rectangle.

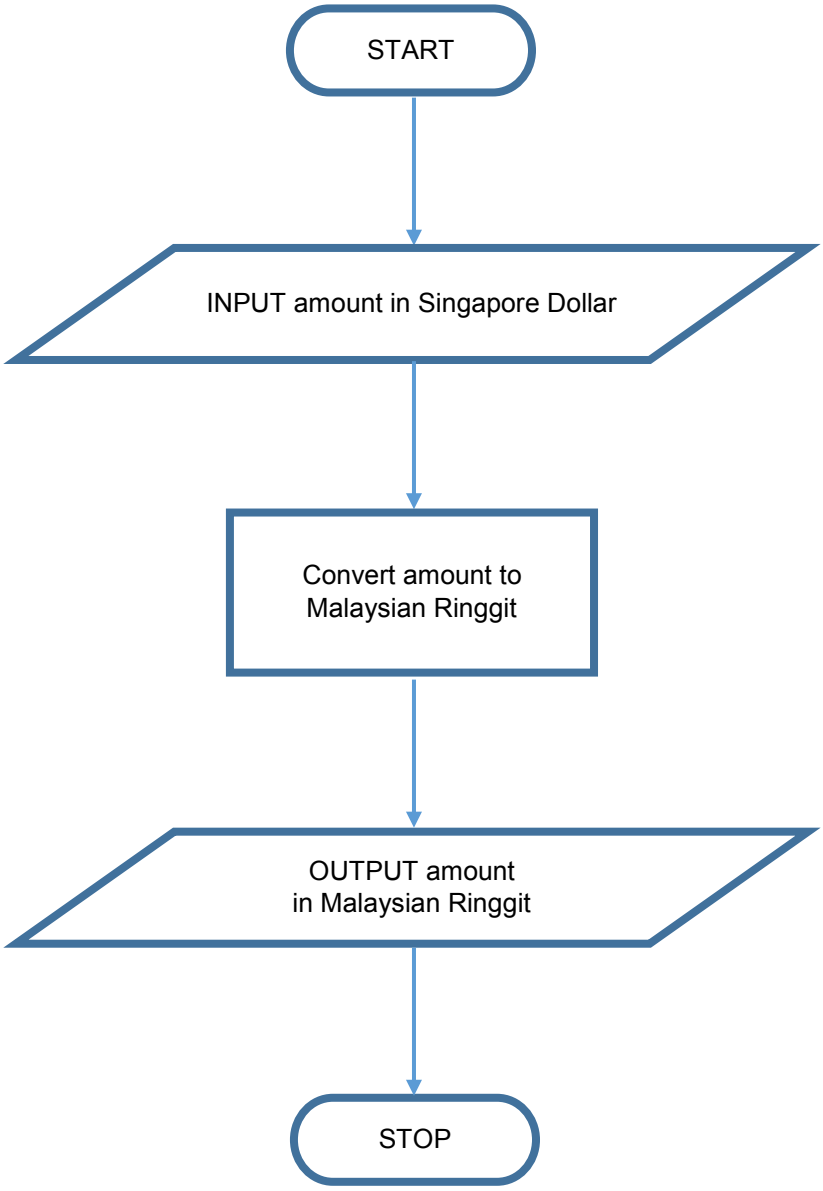
The process symbol is a rectangle. It represents a step involving an operation. This usually involves changing the value of a variable or performing more complex actions.

An example is shown below.

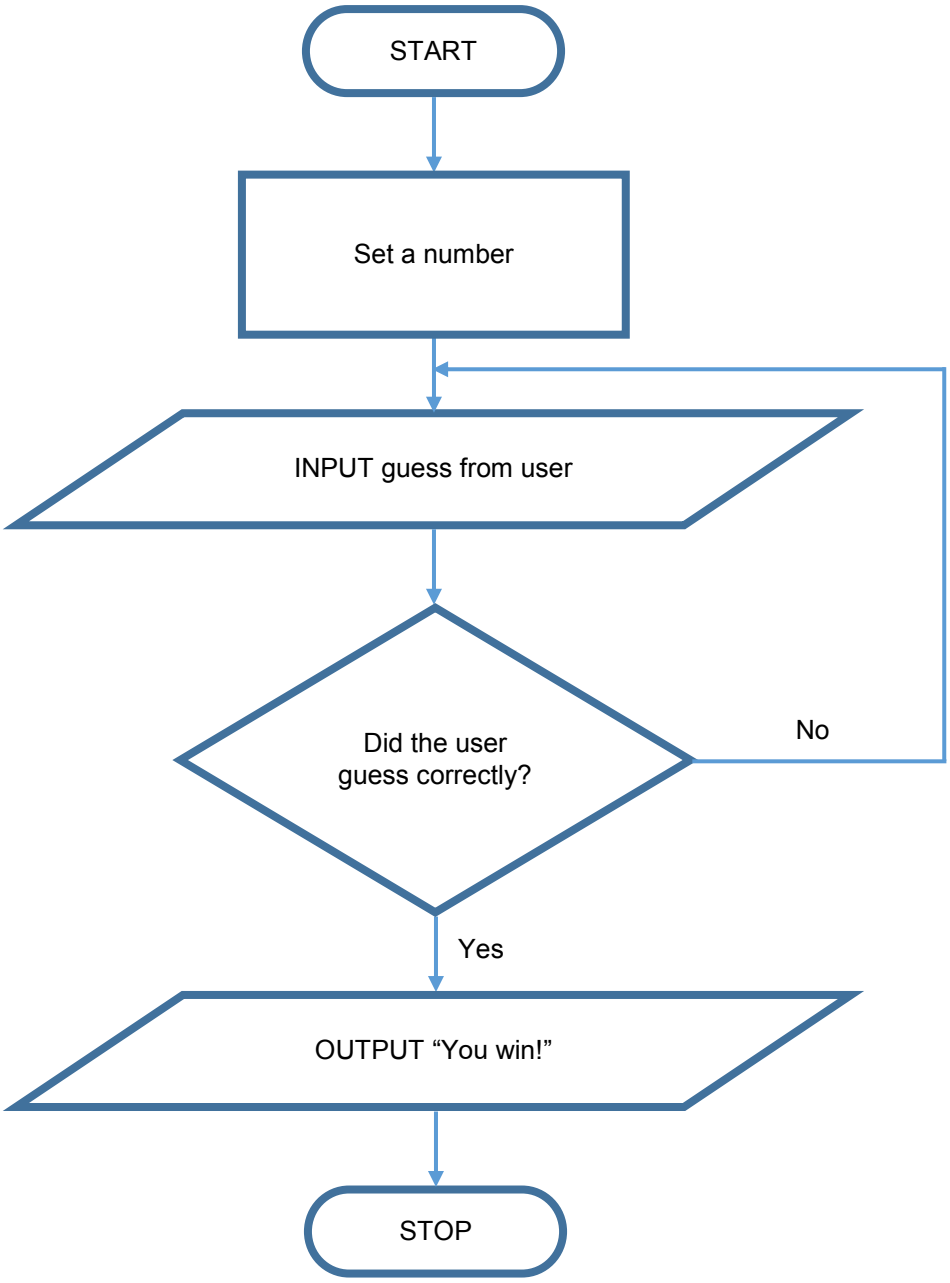


Examples

The example below shows a flowchart of an algorithm to convert from Singapore Dollar to Malaysian Ringgit.



Another example below shows how a guess-the-number game works.

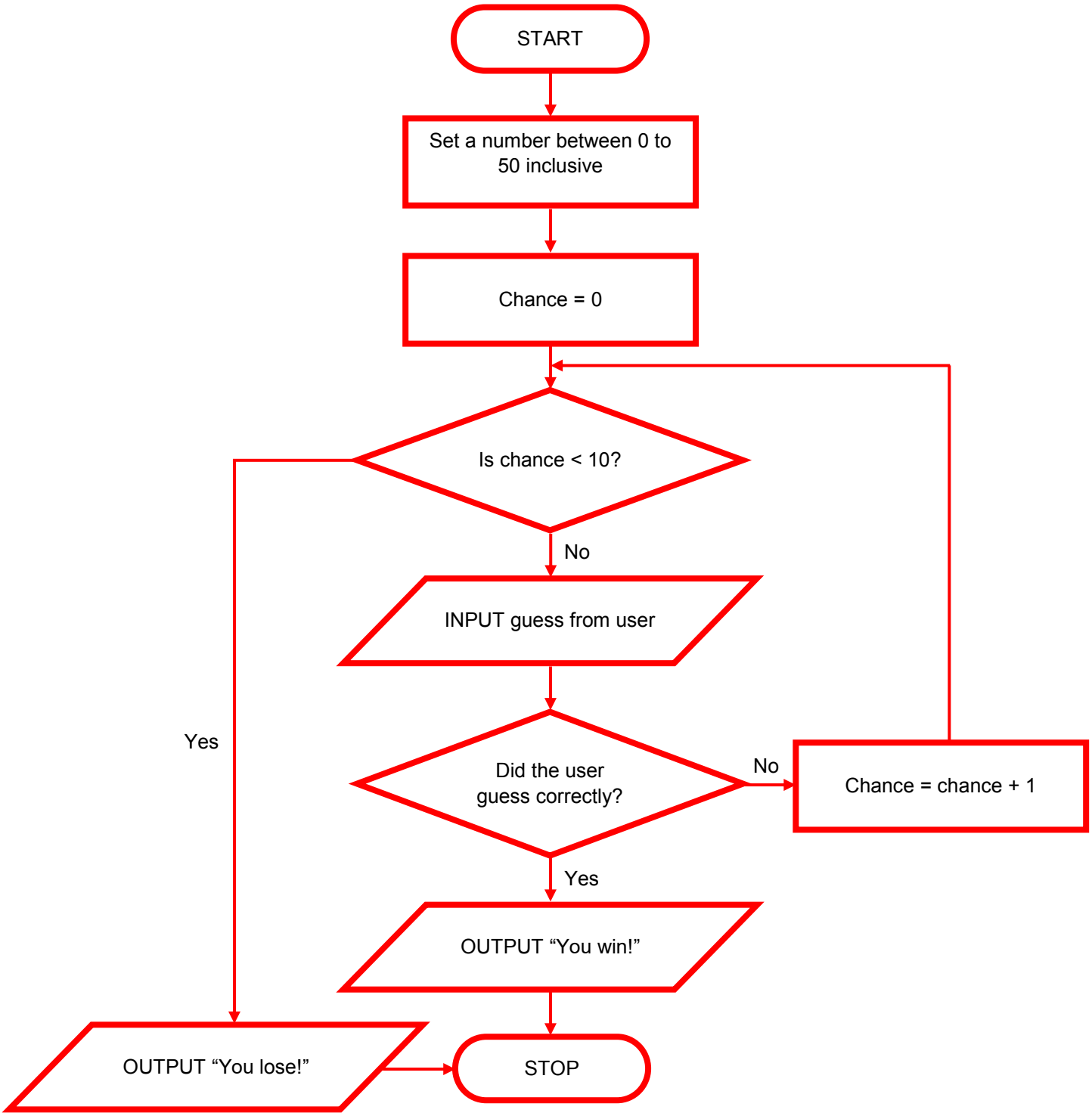


Exercise

Two new rules are to be implemented for the guess-the-number game mentioned in the previous page.

1. The range of numbers is limited to 0 to 50 inclusive.
2. The user has up to 10 chances to guess the number.

Draw a new flowchart to capture the new rules.



Decision Table

Decision table is a visual logic representation for specifying which actions to perform depending on given conditions. Possible combinations of conditions are considered before deciding on the action to be taken.

A useful tool for program testing, it is used to analyse a situation where the conditions and actions involved are more complex.

Shown below is the typical format of a decision table.

Conditions	<Condition 1>	Y	Y	…
	<Condition 2>	Y	N	…
	…	…	…	…
Actions	<Action 1>	X		…
	<Action 2>		X	…
	…			…

Examples

An example below shows a two-condition decision table.

Conditions	>= 70 marks	Y	Y	N	N
	< 45 marks	Y	N	Y	N
Actions	Grade ‘A’	-	X		
	Grade ‘Pass’	-			X
	Grade ‘Fail’	-		X	

Take note that the leftmost scenario cannot take place as a particular score cannot be larger than or equal to 70 marks and less than 45 marks at the same time. As such, we put the dash symbol (-) on each of the actions to indicate the impossibility of such a scenario.

Another example below shows a three-condition decision table.

Conditions	Male	Y	Y	Y	Y	N	N	N	N
	Singaporean or 2nd gen. PR	Y	Y	N	N	Y	Y	N	N
	Healthy	Y	N	Y	N	Y	N	Y	N
Actions	Serve NS	X							
	Do not need to serve NS		X	X	X	X	X	X	X

Notice that some of the cells are actually redundant since NS is mandatory only to those who satisfy all the three conditions specified.

It is possible to come up with a simpler decision table and eventually a simpler program code to be written.

Conditions	Male	Y	N	-	-
	Singaporean or 2nd gen. PR	Y	-	N	-
	Healthy	Y	-	-	N
Actions	Serve NS	X			
	Do not need to serve NS		X	X	X

The dash symbol (-) for the conditions means that they can either be true or false for the given intended actions.

Exercise

An airline company offers discounted tickets according to the following rules:

1. 5% discount applies to every passenger aged 3 and above who purchase a ticket 90 days prior to the departure date.

2. 80% discount applies to every infant aged 0 to 2 occupying a seat.
   No further discount can be granted for an infant ticket.

Draw a decision table for the scenario above.

Conditions	Age <= 2	Y	Y	N	N	N
	Age >= 3	Y	N	Y	Y	N
	Buy 90 days in advance	-	-	Y	N	-
Actions	Offer usual price	-			X	-
	Offer 5% discount	-		X		-
	Offer 80% discount	-	X			-

2020 JC1 H2 Computing 9569

05. Functions

For a start, take a look at the program code below. What is it trying to do?

In [ ]:
```
init1 = 25
result1 = init1 + 273.15
print(result1)

init2 = 30
result2 = init2 + 273.15
print(result2)
```

Notice that the same calculation is performed on *init1* and *init2*. Should we wish to repeatedly perform the same calculation for different values, we can write a **function**, which is a block of organised, reusable codes. It helps us simplify our program and reduce redundancy.

A function has the following format.

In [ ]:
```
def <name_of_function>(<arguments>):
    <do something>
    <return something>
```

For the example above, we can define the following function and subsequently make use of it for different values of *T* passed into the function as an **argument**.

In [ ]:
```
def C_to_K(T_C):
    return T_C + 273.15
```

In [ ]:
```
# Converts 25 degree Celcius to Kelvin
result = C_to_K(init1)
print(result)

# Converts 30 degree Celcius to Kelvin
print(C_to_K(init2))

# Converts 50 degree Celcius to Kelvin
print(C_to_K(50))
```

Notice that with the function *C_to_K* made available to the user, he/she simply needs to call it to do the job without having to know the formula to convert the temperature from degree Celcius to Kelvin. This idea is known as **abstraction**, which is the process of hiding details to reduce complexity.

Defining functions allows us to group multiple steps under a common name, reducing the need to rewrite codes (or reinvent the wheel) if the set of steps needs to be repeated on multiple occasions. It also helps us to break up a big (and difficult) task into multiple smaller (and simpler) parts.

Take note of the following:

- A function need not have an argument or a return statement.
  When a return statement is not supplied, the function returns None.

In [ ]:
```
# Example 1.1
def print_gaps():
    print()
    print()
    print()
```

In [ ]:
```
print("Good morning, world!")
print_gaps()
print("Good afternoon, world!")
print_gaps()
print("Good night, world!")
```

In [ ]:
```
x = print_gaps()
print(x)
```

In [ ]:
```
# Example 1.2
def return_something():
    return "This is something!"
```

In [ ]:
```
text = return_something()
print(text)
```

- A function can have more than one argument, e.g.

In [ ]:
```
# Example 2.1
def greet(name1, name2):
    print("Hello, " + name1 + "!")
    print("Hello, " + name2 + "!")
    print("Nice to meet both of you!")

greet("John", "Jane")
```

In [ ]:
```
# Example 2.2
# Write a function that takes in three numbers and returns their sum.

def add(x, y, z):
    return x + y + z

# Note that sum() is an in-built Python function.
# Should we also use 'sum' as the name of the function in this example, the i
```

In [ ]:
```
print(add(100, 200, 300))
```

- A function can be passed as an argument into another function, i.e. function chaining is possible.

In [ ]:
```
# Example 3.1
# Given the functions C_to_K() and K_to_F, use both of them to print the resu
def K_to_F(T):
    return T * 9/5 - 459.67

print(K_to_F(C_to_K(50)))
```

- A function can be called inside another function.

In [2]:
```
# Example 4.1
def area_of_rectangle(length, breadth):
    return length * breadth

def area_of_n_rectangles(n, length, breadth):
    area = area_of_rectangle(length, breadth)
    return n * area
```

In the example above, we called a function that had been declared previously to help us solve the problem. Of course, we could have written the entire solution without calling other functions.

However, it is a good programming practice to break up a big problem to several small functions. This makes debugging, a chore every programmer hates, much easier.

Exercise

(a) Write a function that takes in an integer and returns a string to tell the user whether it is an odd or an even number.

In [3]:
```
def check_odd_even(x):
    if x % 2 == 0:
        return "It is an even number."
    else:
        return "It is an odd number."
```

(b) Write a function that takes in a string and returns its first character.

In [ ]:
```
def first_char(s):
    return s[0]
```

(c)

The volume of a cuboid is given as follows:

$$volume = length \times breadth \times height$$

The density of an object is given as follows:

$$density = mass / volume$$

(i)
Write a function that takes in the length, breadth and height of a cuboid to calculate its volume and return the value.

In [4]:
```
def volume(l, b, h):
    return l * b * h
```

(ii)
Using the function in **(c)(i)**, write another function that takes in the length, breadth, height and mass of a cuboid. It should calculate and return the density of the cuboid.

In [5]:
```
def density(l, b, h, m):
    vol = volume(l, b, h)
    return m/vol
```

(iii)

Hence, write a program code that:

- asks the user to input four values: length, breadth, height and mass
- display the volume and the density of the cuboid based on the user input

In [9]:
```
l = float(input("Input the length: "))
b = float(input("Input the breadth: "))
h = float(input("Input the height: "))
m = float(input("Input the mass: "))

print("The volume of the cuboid is " + str(volume(l,b,h)))
print("The density of the cuboid is " + str(density(l,b,h,m)))
```

```
Input the length: 1
Input the breadth: 1
Input the height: 1
Input the mass: 1
The volume of the cuboid is 1.0
The density of the cuboid is 1.0
```

2020 JC1 H2 Computing 9569

06. List

List is a collection of data that are ordered and mutable (i.e. changeable), which is denoted by square brackets in Python.

The elements in a list are delimited by commas (,). They can be integers, strings, etc., including other lists, and a list need not contain elements of the same data type.

In [ ]:
```
# Define an empty list
empty_lst = []

# Define some lists with element(s)
fruit = ['apple']
class_1MD10 = ['Sanjay', 'Phoebe', 'Matthew', 'Fauzan']
Matthew_details = ['Matthew', 'M', 17]
```

List Concatenation

It is also possible to merge the contents of two or more lists in order using the addition operator.

In [ ]:
```
lst_a = [1, 2, 3]
lst_b = [4, 5, 6]

lst_a += lst_b
print(lst_a)

lst_c = lst_b * 3
print(lst_c)
```

List Indexing and Slicing

String and list belong to the same sequence data type, they share some similar operations and methods.

However, unlike a string, a list can be changed, making it versatile.

In [ ]:
```
lst1 = ['John', 'M', 18, 'Basketball']
```

In [ ]:
```
# Print the list
print(lst1)
```

What do the following codes do?

In [ ]:
```
print(lst1[0])
```

In [ ]:
```
print(lst1[0][0])
```

In [ ]:
```
print(lst1[:2])
```

In [ ]:
```
lst1[-1] = 18
print(lst1)
```

Does the following code work?

In [ ]:
```
text = "I am 5 years old."
text[5] = 6
```

Common List Functions and Methods

Function or Method	Description	Example	Output
len(s)	Returns the length of the list	L = [0, 1, 2, 3, 4] len(L)	5
L.pop(i)	Removes the item at given index and return it. If no index is specified, the last item of the list is removed and returned.	L = [0, 1, 2, 3, 4] L.pop() L = ['a', 'b', 'c', 'd'] L.pop(2)	4 'c'

L.append(x)	Appends x to the end of the list	L = [0, 1, 2, 3, 4]
		L.append(5)
			#Returns None, but
			L = [0, 1, 2, 3, 4, 5]

L.sort()	Sorts the list in ascending order	L = [1,5,3,7,2,8]
		L.sort()
			#Returns None, but
			L = [1, 2, 3, 5, 7, 8]

L.remove(x)	Removes x from the list.
	Returns an error if x does not exist.
		L = ['a', 'b', 'c', 'd']
		L.remove('b')
			#Returns None, but
			L = ['a', 'c', 'd']

L.extend(x)	Adds items in the second list to the end of the first list.	L = ['a', 'b', 'c', 'd']
		L.extend('edf')
			#Returns None, but
			L = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

L.clear(x)	Removes all items from the list.	L = ['a', 'b', 'c', 'd']
		L.clear()
			#Returns None, but
			L = []

L.copy()	Creates a copy of the list.	L = ['a', 'b', 'c', 'd']
		L2 = L.copy()
			#Returns None, but
			L2= ['a', 'b', 'c', 'd']

L.index(x)	Returns first index of given value in the list or raises ValueError if given value is not found	L = ['a', 'b', 'c', 'd'] L.index('b')	1

L.insert(index, item)	Inserts at given index the given item	L = ['a', 'b', 'c', 'd']
		L.insert(2, 'x')
			#Returns None, but
			L= ['a', 'b', 'x', 'c', 'd']

L.reverse()	Reverses list in-place	L = ['a', 'b', 'c', 'd']
		L.reverse()
			#Returns None, but
			L= ['d', 'c', 'b', 'a']

L.sort()	Sorts list in-place	L = [1,3,2,9,8,7,6,4,5]
		L.sort()
			#Returns None, but
			L= [1, 2, 3, 4, 5, 6, 7, 8, 9]

You can use the space below to experiment with the different functions and methods.

In [ ]:

Appendix

There are other operations and methods that we can perform on lists.

Refer to the official documentation for a more complete list.

https://docs.python.org/3.6/library/stdtypes.html#mutable-sequence-types

2020 JC1 H2 Computing 9569

07. Number Bases

Underlying our sophisticated computer systems and applications are values stored as a series of 0s and 1s. Computers perform operations using only these two digits as the arithmetic encoded in the hardware follows a **binary (base 2)** number system instead of the **denary (base 10)** one that we use in our daily lives.

Denary (Base 10)

To understand how number bases work, it is helpful to take a closer look at our denary number system and see how numbers are represented in Base 10. Note that we use the digits 0 to 9 as we do not require a number larger than 9 to be represented by a single symbol.

Example 1

345 actually represents the number $3 \times 10^2 + 4 \times 10 + 5 \times 1$.

We may also represent it in tabular form as shown below:

10^2	10^1	10^0
3	4	5

We can also begin from the right and multiply the first (rightmost) digit, 5, by $10^0 = 1$, followed by the next digit, 4, by 10^1 and so on, until we reach the last (leftmost) digit.

Example 2

142,057 actually represents the number
$1 \times 10^5 + 4 \times 10^4 + 2 \times 10^3 + 0 \times 10^2 + 5 \times 10 + 7 \times 1$.

Example 3

The number system also extends to decimals, where the first digit to the right of the decimal point is multiplied by 10^{-1}, the second digit by 10^{-2}, and so on, until we reach the last digit.

3.14 actually represents the number $3 \times 1 + 1 \times 10^{-1} + 4 \times 10^{-2}$.

Think!

- Which fractions have finite decimal representations, i.e. they do not go on forever?

Binary (Base 2)

In Base 2, we do not require numbers larger than 1 to be represented by a single symbol. Therefore, we only use the digits 0 and 1. The notation works in the same way as Base 10 notation, but instead of multiplying the digits by powers of 10, we multiply them by powers of 2.

Example 4

The binary number 1110 is actually $1 \times 2^3 + 1 \times 2^2 + 1 \times 2 + 0 \times 1$ (which gives us 14 in denary notation).

2^3	2^2	2^1	2^0
1	1	1	0

When we are talking about numbers from two different bases at the same time, it is helpful to use a subscript to represent the base. Therefore, we would write that $1110_2 = 14_{10}$.

Try writing the numbers 1_{10} to 10_{10} in binary.

Think!

- Which numbers end in a 0 in their binary representation? Which numbers end in a 1?
- What happens when you multiply a binary number by 2_{10}?

Example 5

To convert the denary number 92 into binary, we need to express it as a sum of powers of 2.

$$
\begin{aligned}
&92_{10} = 64_{10} + 16_{10} + 8_{10} + 4_{10} \\
&= 2^6 + 2^4 + 2^3 + 2^2 \\
&= 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 0 \times 1 \\
&= 1011100_2
\end{aligned}
$$

As writing numbers in binary can quickly become very long, we often abbreviate them using **octal (Base 8)** and **hexadecimal (Base 16)** number systems.

Octal (Base 8)

In Base 8, we do not require numbers larger than 7 to be represented by a single symbol. Therefore, we only use the digits 0 to 7, and we multiply them by powers of 8.

Example 6

The octal number 157_8 is actually $1 \times 8^2 + 5 \times 8 + 7 \times 1 = 111_{10}$ in denary.

Think!

- Here is a joke. Why do computer programmers often confuse Halloween and Christmas? (Hint: Halloween is on Oct 31, while Christmas is on Dec 25.)

- Is there a way to convert between binary and octal very quickly without a computer or calculator?

Hexadecimal (Base 16)

In Base 16, we need to be able to represent the numbers 0 to 15_{10} by a single symbol each. We use the digits 0 to 9, and then the letters A to F to represent 10_{10} to 15_{10}.

Example 7

The hexadecimal number B07A is actually
$11 \times 16^3 + 0 \times 16^2 + 7 \times 16 + 10 \times 1 = 45,178_{10}$

Example 8

To convert the denary number 1977 into hexadecimal, we need to express it terms of powers of 16.
$1977_{10} = 7 \times 16^2 + 11 \times 16 + 9 \times 1 = 7\mathrm{B}9$ in hexadecimal.

Think!

- Is there a way to convert between binary and hexadecimal very quickly without a computer or calculator?

Common Uses of Number Bases

Number base	Use
Binary	Machine-level computation, Boolean (True or False) conditions
Octal	Older 12/24/36-bit systems
Denary	Display for easy human readability
Hexadecimal	Modern 16/32/64-bit systems

2020 JC1 H2 Computing 9569

08. Iteration

Consider a situation where we want to display positive integers from 1 to 5 line by line. We may instinctively think of doing the following.

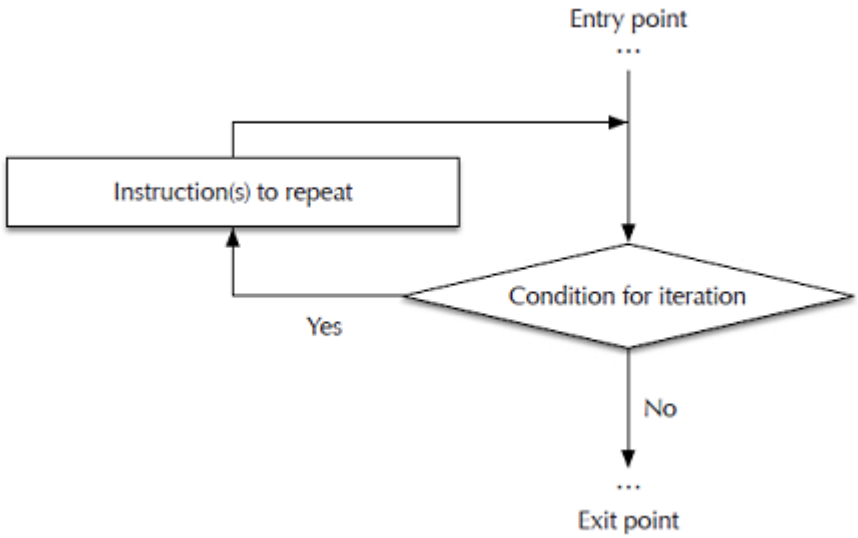
In [ ]:
```
print(1)
print(2)
print(3)
print(4)
print(5)
```

What if we need to go all the way to 100? Surely there must be a more efficient way to do so.

Iteration refers to the process of repeating a task to achieve a specific end goal.

When iteration is employed, the set of instructions within the **loop** will keep repeating until a certain condition has been reached or is no longer satisfied. The result(s) of one iteration is/are used as the starting point for the next iteration.

The flowchart diagram below shows how iteration works.



FOR Loop

FOR loop is a looping mechanism that comes with an explicit counter for every iteration, executing a given task until the counter reaches a certain value.

Before we discuss the construct of the FOR loop, let us take a look at the **range()** method that is used in conjuction with the loop. This function returns a range object comprising a sequence of integers.

In [ ]:
```
range(<start>, <stop>, <step>)
```

Does the above construct look somewhat familiar to you?

Start is an optional argument that determines the first number in the sequence. If not supplied, the value is 0 by default.

Stop is a mandatory argument that determines the last number in the sequence, which is *stop*-1.

Step is an optional argument that determines the increment or decrement of the numbers. If not supplied, the value is 1 by default.

For visualisation, we can typecast a range object into a list as shown in the examples below.

In [ ]:
```
# Print a list of integers from 0 to 4
a = range(5)
print("a is", list(a))

# Print a list of integers from 1 to 4
b = range(1,5)
print("b is", list(b))

# Print a list of odd integers from 1 to 10
c = range(1, 10, 2)
print("c is", list(c))

# Print a list of integers from 3 to 0
d = range(3, -1, -1)
print("d is", list(d))
```

Once we have understood how the method works, we shall look into the FOR loop construct given below.

In [ ]:
```
for i in range(<start>, <stop>, <step>):
    <do something>
```

In the above construct, the variable *i* represents the counter with an initial value of *start* (or 0 if *start* is not supplied).

The iteration continues until *i >= stop-1*.

Take note that the body, which contains the task(s) to be performed at every iteration, has to be indented.

Examples

The code below prints out the integers 0 to 5.

In [ ]:
```
for i in range(6):
    print(i)
```

The trace table below shows what happens at every iteration.

Iteration	i	Execute
1	0	print(0)
2	1	print(1)
3	2	print(2)
4	3	print(3)
5	4	print(4)

6	5	print(5)
7	6	Exit Loop

The code below prints out the integers 1, 3, 5 and 7.

In [ ]:
```
for i in range(1, 8, 2):
    print(i)
```

The trace table below shows what happens at every iteration.

Iteration	i	Execute
1	1	print(1)
2	3	print(3)
3	5	print(5)
4	7	print(7)
5	9	Exit Loop

WHILE Loop

WHILE loop is a looping mechanism that continually executes a given task while a particular condition evaluates to True.

In [ ]:
```
while (<condition>):
    <do something>
```

Similar to FOR loop, the body comprising the task(s) to be performed at every iteration has to be indented.

We need to be especially careful when using a WHILE loop. There must be a point in time where the condition has to evaluate to False, otherwise we will end up in an infinite loop.

See what happens when you run the code below, where *i* is always equal to zero and thus the condition stated in the loop always evaluates to True. Do interrupt the kernel after looking at the output.

In [ ]:
```
i = 0

while (i < 3):
    print("Hello!")
```

To use a counter for a WHILE loop, it is declared first outside the loop and subsequently incremented inside the loop.

Examples

The code below prints out the integers 0 to 5.

In [ ]:
```
i = 0

while (i < 6):
    print(i)
    i += 1
```

The trace table below shows what happens at every iteration.

Iteration	i	i < 6	Execute
1	0	True	print(0) i+=1
2	1	True	print(1) i+=1
3	2	True	print(2) i+=1
4	3	True	print(3) i+=1
5	4	True	print(4) i+=1
6	5	True	print(5) i+=1
7	6	False	Exit Loop

The code below prints out the integers 1, 3, 5 and 7.

In [ ]:
```
i = 1

while (i < 8):
    print(i)
    i += 2
```

The trace table below shows what happens at every iteration.

Iteration	i	i < 8	Execute
1	1	True	print(1) i+=2
2	3	True	print(3) i+=2
3	5	True	print(5) i+=2
4	7	True	print(7) i+=2
5	9	False	Exit Loop

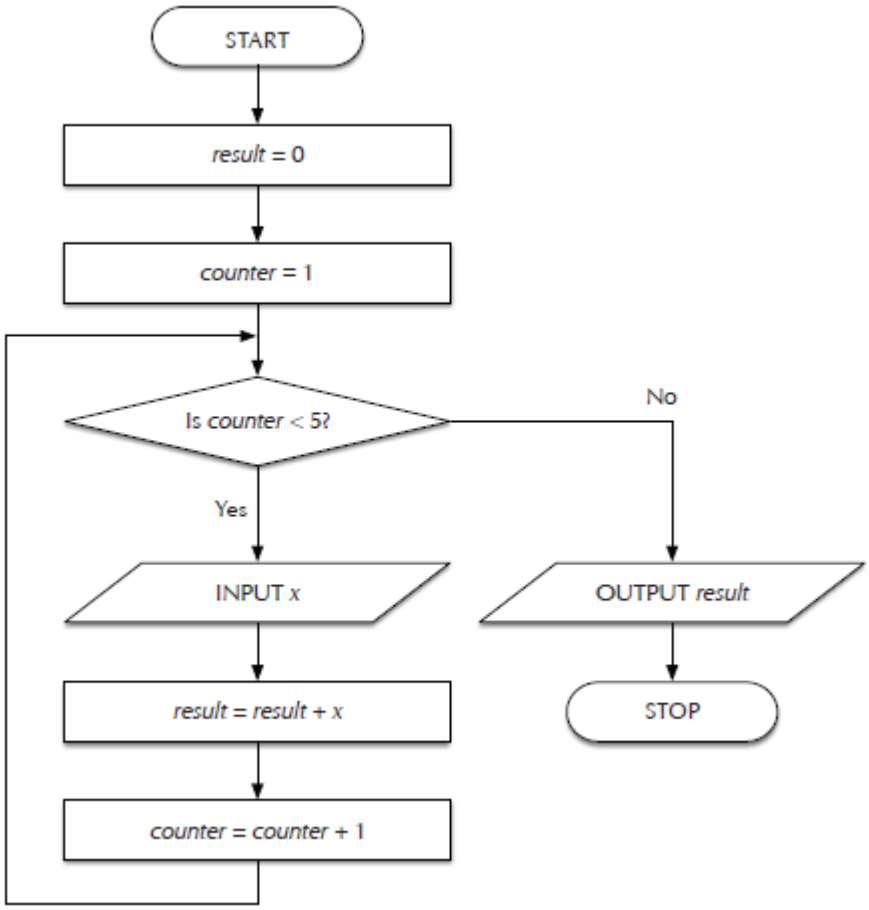
Exercise

Write two separate program codes using a FOR loop and a WHILE loop respectively to print out the integers 5, 4, 3, 2 and 1 in descending order.

In [ ]:
```
for i in range(5, 0, -1):
    print(i)
```

In [ ]:
```
i = 5

while i > 0:
    print(i)
    i -= 1
```

Let us now take a look at the following flowchart diagram that shows how we can count the sum of four numbers keyed in by a user.



We shall write two separate program codes that displays the following upon execution.

```
Input an integer: 1
Input an integer: 2
Input an integer: 3
Input an integer: 4
Sum = 10
```

Write a code that makes use of FOR loop.

In [ ]:
```
result = 0

for counter in range(1, 5):
    x = input("Input an integer: ")
    result += int(x)

print("Sum = " + str(result))
```

Write a code that makes use of WHILE loop.

In [ ]:
```
result, counter = 0, 1

while counter < 5:
    x = input("Input an integer: ")
    result += int(x)
    counter += 1

print("Sum = " + str(result))
```

Nested Loop

Just like the IF statement, we can have loop(s) nested inside another.

Take a look at the two examples shown below. Try to guess the output before running the code.

In [ ]:
```
result = 0

for i in range(3):
    result += i
    print(result)
    for j in range(2):
        print("Hello!")
```

In [ ]:
```
j = 0

for i in range(3):
    print("Here we go!")
    j=0
    while (j < 3):
        print(j)
        j += 1
```

Break and Continue

These are two keywords that may be used with loops.

To understand the difference between the two, let us try to print out integers from 10 to 15, but we want to avoid the unlucky number 13.

In [ ]:
```
for i in range(10, 15):
    if (i == 13):
        break

    print(i)

print("Done!")
```

In [ ]:
```
for i in range(10, 15):
    if (i == 13):
        continue

    print(i)

print("Done!")
```

It should not be difficult to see that **break** causes the program to skip the remaining code inside the loop and exit the loop completely.

On the other hand, **continue** causes the program to skip the remaining code inside the loop for that particular iteration and continue with the next iteration, if any.

Loops for Iterables

Strings, lists and tuples (to be covered in the next topic) are examples of data types known as iterables.

FOR loop, in particular, is commonly used to easily iterate over members of an iterable.

String

We have previously learnt that a string is a sequence of characters. In order for us to extract and print out each character line by line, we can do the following.

In [ ]:
```
s = "Hello"

for i in s:
    print(i)
```

The trace table below shows what happens at every iteration.

Iteration	i	Execute
1	'H'	print('H')
2	'e'	print('e')
3	'l'	print('l')
4	'l'	print('l')
5	'o'	print('o')

Alternatively, since each character of a string has an index, we can also use the range() function introduced earlier to get the job done.

The following code prints out each character three times line by line.

In [ ]:
```
s = "Hello"

for i in range(len(s)):
    print(s[i] * 3)
```

The trace table below shows what happens at every iteration.

Iteration	i	Execute
1	0	print(s[0]*3)
2	1	print(s[1]*3)
3	2	print(s[2]*3)
4	3	print(s[3]*3)
5	4	print(s[4]*3)

Exercise

Write a code to count the number of 'o' in the string given below. Print the number as an output.

In [ ]:
```
s = "I love Python."

count = 0

for i in s:
    if i == 'o':
        count += 1

print(count)

# OR

count = 0
```

```
for i in range(len(s)):
    if s[i] == 'o':
        count += 1

print(count)
```

List

Recall that a list is an ordered sequence of elements. In a similar fashion to strings, we can extract and print out each element line by line.

In [ ]:
```
l = ['apple', 'banana', 'cherry', 'durian']

for i in l:
    print(i)
```

The trace table below shows what happens at every iteration.

Iteration	i	Execute
1	'apple'	print('apple')
2	'banana'	print('banana')
3	'cherry'	print('cherry')
4	'durian'	print('durian')

Since a list is mutable, it is possible for us to modify its elements. This can be done when we use the range() function to deal with the indexes of the elements.

Given the same list of strings, we can change all of them to upper case as follows.

In [ ]:
```
l = ['apple', 'banana', 'cherry', 'durian']

for i in range(len(l)):
    l[i] = l[i].upper()
```

The trace table below shows what happens at every iteration.

Iteration	i	Execute
1	0	l[0] = l[0].upper()
2	1	l[1] = l[1].upper()
3	2	l[2] = l[2].upper()
4	3	l[3] = l[3].upper()

Exercise

Given a list of positive integers below, modify it such that the even integers are multiplied by 2.

In [ ]:
```
l = [1, 5, 2, 3, 6, 9, 8, 4, 7]

for i in range(len(l)):
    if l[i]%2 == 0:
        l[i] *= 2
```

```
print(l)
```

Once you have successfully completed the exercise above, copy and paste your code to the box below.

Modify your code such that the elements in *l1* are copied to *l2* with the even integers multiplied by 2. Do not change the contents of *l1*.

In [ ]:
```
l1 = [1, 5, 2, 3, 6, 9, 8, 4, 7]
l2 = []

for i in range(len(l1)):
    if l1[i]%2 == 0:
        l2.append(l1[i]*2)
    else:
        l2.append(l1[i])

print(l1)
print(l2)
```

Write the function *vowel_counter* that takes in a list and counts the number of vowels in a list of words. It should return an integer.

e.g.

```
lst = ["hello", "bye"]
vowel_counter(lst) --> should return 3.
```

In [ ]:
```
lst = ["apple", "banana", "cherry", "durian"]

def vowel_counter(lst):
    count = 0
    for word in lst:
        for char in word:
            if char in "aeiou":
                count += 1
    return count

print(vowel_counter(lst)) # You should obtain 9.
```

Common Mistakes

Be wary when using FOR loops for iterables when inserting or deleting elements.

Case 1

The code below tries to do two things:

1. insert 'x' into the list after 'b', and
2. print out all the items in the list one by one.

While the code below looks reasonable at first glance, notice that it does not behave as expected.

In [ ]:
```
lst1 = ['y', 'b', 'k', 'j', 'o']

for i in range(len(lst1)):
    if (lst1[i] == 'b'):
```

```
            lst1.insert(i+1, 'x')
        print(lst1[i])
```

Observe that while the inserted letter 'x' is printed out, the letter 'o' is not. That is due to the fact that the integer *i* only runs up to the original length of the list, which is from 0 to 5, and it is not modified despite the changing length of the list.

Case 2

Let us now take a look at the case of deletion. Given another list of letters, suppose we want to print out all the letters in order one by one and remove the letter 'c' as well. Observe what happens when you run the following code.

In [ ]:
```
lst2 = ['m', 'p', 'c', 'f', 'v']

for i in range(len(lst2)):
    print(lst2[i])
    if (lst2[i] == 'c'):
        lst2.pop(i)
```

What error message does the code give when it is run? Similar to the previous case, the integer *i* runs up to the original length of the list, which is from 0 to 5, and is not affected by the change in the length of the list.

Exercise

Rewrite the program code for each of the cases shown above using WHILE loop.

In [ ]:
```
lst1 = ['y', 'b', 'k', 'j', 'o']

i = 0

while i < len(lst1):
    if (lst1[i] == 'b'):
        lst1.insert(i+1, 'x')
    print(lst1[i])
    i += 1
```

In [ ]:
```
lst2 = ['m', 'p', 'c', 'f', 'v']

i = 0

while i < len(lst2):
    print(lst2[i])
    if (lst2[i] == 'c'):
        lst2.pop(i)
    else:
        i += 1
```

2020 JC1 H2 Computing 9569

09. Tuple

Tuple is a collection of data that are ordered and immutable (i.e. unchangeable), which is denoted by round brackets in Python.

Similar to the case of list, the elements in a tuple are delimited by commas (,). They can be integers, strings, etc., including other tuples, and a tuple need not contain elements of the same data type.

In [ ]:
```
# Define an empty tuple
tup_a = ()

# Define a tuple with only one element
tup_b = (100,)

# What about this?
tup_c = (100)
print(tup_c)
print(type(tup_c))

# Define a tuple with more than one element
tup_d = (1, 2, 3, "Happy new year!")
```

Tuple Indexing and Slicing

Try guessing the outputs when you perform the following operations before checking your answers.

- tup[6]
- tup[-1][1]
- tup[0:2]

In [ ]:
```
tup = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100, (110, 120, 130))
```

In [ ]:
```
print(tup[6])
print(tup[-1][1])
print(tup[0:2])
```

Immutability

As mentioned earlier, since tuples are immutable, it is not possible to add, remove or edit any elements associated with them. Try executing the code below and see what happens.

In [ ]:
```
tup1 = (10, 20, 30, 40, 50)
tup1[-1] = 500
```

However, using the addition and multiplication operators on tuples may work.

In [ ]:
```
tup2 = (1, 2, 3)
tup3 = (4, 5, 6)
```

In [ ]:
```
# Is this allowed?
```

```
tup4 = tup2 + tup3
print(tup4)
```

In [ ]:
```
# Is this allowed?
tup5 = tup2 * 3
print(tup5)
```

By adding *tup2* with *tup3*, the two tuples remain as they are. A new tuple, *tup4*, is created in the memory space with all the elements of *tup2* and *tup3* in order.

Similarly, in multiplying *tup2* by a factor of 3, *tup2* remains as it is. A new tuple, *tup5*, is created in the memory space with a total of 9 elements.

In [ ]:
```
# Is this allowed?
tup4 += tup5
print(tup4)
```

In essence, it is impossible to modify a tuple. We can, however, let it point to another set of values.

In [ ]:
```
tup5 = ('Computing', 'is', 'fun')
print(tup5)
```

Common Tuple Functions and Methods

Most of the methods available to lists do not work for tuples since tuples are immutable. However, the following methods do work.

You can use the space below to experiment with the different functions and methods.

In [ ]:
```

```

Iterating Through Tuples

Since a tuple is an iterable, we can use a loop to iterate through its elements.

Recapping on what we have learnt from the previous topic, here are two ways we can print out all the elements in a tuple line by line.

In [ ]:
```
tup8 = ('Alan', 'Betty', 'Charlie', 'Diana', 'Ethan')

for item in tup8:
    print(item)
```

In [ ]:
```
for i in range(len(tup8)):
    print(tup8[i])
```

In the example below, the function takes in a tuple and returns a tuple containing all numbers that are divisible by 9.

In [ ]:
```
def div_3(tup):
    ans = ()
    for item in tup:
        if item % 3 == 0:
            ans += (item,)
```

```
    return ans

print(div_3((1,2,3,4,5,6,7,8,9)))
```

Exercise

Part of the function *remove_string*, which takes in a tuple, is given below. Complete the program code such that the function returns a new tuple with elements that are not strings.

In [ ]:
```
def remove_string(tup):
    ans = ()
    for item in tup:
        if type(item) != str:
            ans += (item,)
    return ans

print(remove_string(('a',1,2,3,'b',4,6,7,'c',True)))

# Additional example: removing not only strings, but also
boolean
def remove_str_bool(tup):
    ans = ()
    for item in tup:
        if type(item) not in (str, bool):     # OR if
type(item) != str and type(item) != bool:
            ans += (item,)
    return ans

print(remove_str_bool(('a',1,2,3,'b',4,6,7,'c',True,8.0,9.0)))
```

Write the function *odd_tuple* that takes in a tuple and returns a new tuple with only the odd numbers.

In [ ]:
```
def odd_tuple(tup):
    result = ()
    for item in tup:
        if item%2 == 1:
            result += (item,)
    return result

# Test your code with the following.
# odd_tuple((1,2,3,4,5)) --> (1, 3, 5)
# odd_tuple((2,4,6,8)) --> ()
# odd_tuple((2,4,6,8,9)) --> (9,)
```

Write the function *string_index* that takes in a tuple and returns a new tuple with only the indexes of the strings.

In [ ]:
```
def string_index(tup):
    ans = ()
    for i in tup:
        if type(i) == str:
            ans += (tup.index(i),)
    return ans

# Test your code with the following.
# string_index((1,2,3,'four',5,'six')) --> (3, 5)
# string_index(('happy','sad','joy',2,3,4)) --> (0, 1, 2)
```

In [ ]:

Typecasting

Note that we can typecast a string into a tuple or a list.

In [ ]:
```
text = "abc"
print(tuple(text))
print(list(text))
```

A tuple can be typecast into a list and vice versa.

In [ ]:
```
tup7 = tuple(['a','b','c'])
print(tup7)
print(list(tup7))
```

Use of Tuples

At this juncture, we may feel that tuples are nothing but 'underpowered lists'. Is there any particular reason as to why we employ the use of tuples instead of lists?

One reason is that it makes our codes safer. Using a tuple instead of a list is akin to implying that the data stored are constants. As a simple example, think about the colours of the rainbow. As we all know, the seven colours are (from the longest to the shortest wavelength): red, orange, yellow, blue, green, indigo and violet.

Storing the colours of the rainbow as strings in a tuple should prevent any changes to them. Using a list, on the other hand, allows one to change the element and subsequently compromise the concept discussed at hand.

In [ ]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue',
'indigo', 'violet']

rainbow[-1] = 'hot pink'

print("The last colour of the rainbow is " + rainbow[-1] + ".")
```

Not In Syllabus: Filter and Map

Filter

As the name suggests, **filter** removes some elements from a tuple or a list. It creates a new **filter object** containing only elements for which a specified function returns True. This object needs to be typecasted to a tuple or a list for further use.

Filter takes in two arguments: a boolean function and an iterable data type.

In [ ]:
```
filter(name_of_function, iterable_data_type)
```

Refer to the example below, which only takes in odd integers from the input tuple into a filter object, *odd_num1*. This object is then typecasted to a tuple,

odd_num2, for display.

In [ ]:
```
def check_odd(x):
    return x%2 == 1

numbers = (48, 31, 77, 100, 95, 2, 0, 13, 209)

result1 = filter(check_odd, numbers)
result2 = tuple(result1)

print(result1)
print(result2)
```

Map

Map applies a function to all the elements in a tuple or a list into a new **map object** as the output. Similarly, this object needs to be typecasted to a tuple or a list for further use.

Map takes in two arguments: a function and an iterable data type.

In [ ]:
```
map(name_of_function, iterable_data_type)
```

Refer to the example below, which multiplies all the elements in the input tuple by a factor of 5 into a map object, *result1*. This object is then typecasted to a list, *result2*, for display.

In [ ]:
```
def multiply_by_5(x):
    return x*5

numbers = (1, 2, 3, 4, 5, 6)

result1 = map(multiply_by_5, numbers)
result2 = list(result1)

print(result1)
print(result2)
```

Putting Them Together

Suppose we want to find the perfect squares of integers 1 to 10 that are even. We can use both filter and map to do so as shown below.

In [ ]:
```
def square(x):
    return x**2

def check_even(x):
    return x%2 == 0

numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

perfect_squares = tuple(map(square, numbers))
even_perfect_squares = tuple(filter(check_even,
perfect_squares))

print(even_perfect_squares)
```

Function or Method	Description	Example	Output

len(tup)	Returns the length of the tuple	tup = (0, 1, 2, 3, 4) len(tup)	5
tup.count()	Returns the number of times an item occurs in the tuple	tup = (0, 1, 2, 2, 2, 3, 4, 2) tup.count(2)	4
tup.index()	Returns the index of the first occurence for an item in the tuple	tup = (0, 3, 2, 3, 4, 3, 4) tup.index(3)	1
max(tup)	Returns the item with the maximum value in the tuple	tup = (10, 4, 12, 3) max(tup)	12
min(tup)	Returns the item with the minimum value in the tuple	tup = (10, 4, 12, 3) min(tup)	3

2020 JC1 H2 Computing 9569

10. File Input/Output

Python allows us to open and read data from files, as well as create files and write data to them.

For the purpose of the syllabus, we will be working with text files (.txt) most of the time.

Opening and Closing Files

The **open()** function is used to open a file. The format is as follows.

In [ ]:
```
open("sample1.txt", "r") # OR open("sample1.txt")
```

It requires at least one argument, *file_name*, which is the name of the file.

The optional second argument, *file_mode*, is supplied according to what the user intends to do with the file after opening it.

Character	Mode	Function
"r"	Read	Opens a file for reading (throws an error if the file does not exist)
"w"	Write	Opens a file for writing (creates the file if it does not exist)
"a"	Append	Opens a file for appending (creates the file if it does not exist)
"x"	Create	Creates the specified file (throws an error if the file already exists)

If no *file_mode* is supplied, it is in the read mode by default.

It is always a good practice to close any files using the **.close()** method when we are done using them. Very often, changes made to a file may not appear until we close it.

In [ ]:
```
f1 = open("sample1.txt")

# Alternatively,
# f1 = open("sample1.txt", "r")

f1.close()
```

Reading Files

The **.read()** method is used to read the entire content of an opened file.

In [ ]:
```
f1 = open("sample1.txt", "r")

line = f1.read()
print(line)

f1.close()
```

The **.readline()** method, one the other hand, is used to read one line of the file at a time.

In [ ]:
```
f1 = open("sample1.txt", "r")
```

```
line = f1.readline()
print(line)
```

Notice that when the method is executed the second time, the second line of the file is read. The method works as a cursor.

In [ ]:
```
line = f1.readline()
print(line)
```

In [ ]:
```
line = f1.readline()
print(line)
```

In [ ]:
```
line = f1.readline()
print(line)

f1.close()
```

A loop can be used to read the contents of a file.

In [ ]:
```
f1 = open("sample1.txt", "r")

for line in f1:
    print(line)

f1.close()
```

Exercise

Write a program code that:

- reads an input file consisting of numbers;
- stores the numbers in a list;
- adds the numbers together and display the result.

In [ ]:
```
f1 = open("sample2.txt", "r")
lst = []

for line in f1:
    lst.append(int(line))

print(sum(lst))

f1.close()
```

Write the function *capital_count* that takes in an input file of words and returns the number of words that starts with a capital letter.

In [ ]:
```
def capital_count(filename):
    f1 = open(filename)
    count = 0

    for line in f1:
        if line[0].isupper():
            count += 1

    f1.close()

    return count

print(capital_count('sample3.txt'))
```

Writing Files

Writing to an output file is just as easy as reading lines from an input file.

The **.write()** method requires a string as an argument and writes it as a line into an opened file. If the file does not exist, it will be created.

In [ ]:
```
f1 = open("sample4.txt", "w")

f1.write("I just wrote something 1 \n")
f1.write("I just wrote something 2 \n")
f1.write("I just wrote something 3 \n")

f1.close()
```

Try running the code below. Notice that the content of *sample4.txt* will be overwritten.

In [ ]:
```
f1 = open("sample4.txt", "w")

f1.write("I just wrote something 4 \n")
f1.write("I just wrote something 5 \n")
f1.write("I just wrote something 6 \n")

f1.close()
```

The append mode of the **open()** function allows us to add lines into an existing file instead of overwriting its content.

In [ ]:
```
f1 = open("sample4.txt", "a")

f1.write("I just wrote something 7 \n")
f1.write("I just wrote something 8 \n")
f1.write("I just wrote something 9 \n")

f1.close()
```

Exercise

Write the function *write_multiply* that:

- takes in and reads an input file consisting of numbers;
- multiplies the numbers together;
- store the result in a text file named *output.txt*.

Use *sample5.txt* for this question.

In [ ]:
```
def write_multiply(filename):
    f1 = open(filename)
    result = 1

    for line in f1:
        result = result * int(line) # OR result *= int(line)

    f2 = open("output.txt", "w")
    f2.write(str(result))

    f1.close()
    f2.close()

write_multiply("sample5.txt")
```

.split() Method

So far, we have been dealing with data that are only separated by lines in text files. What if they are separated by white spaces (' '), commas (',') or any other characters?

The **.split()** method of a string can be used conveniently to extract the data separated by a particular character into a list. When no argument is passed into the method, the separator is a white space by default.

Refer to the examples below.

In [ ]:
```
row1 = "1 2 3 4 5"
output1 = row1.split()
print(output1)

row2 = "6,7,8,9,10"
output2 = row2.split(",")
print(output2)
```

To ensure that you understand how the method works, why not predict the output of the following?

In [ ]:
```
row3 = "1,2,3,4,5"
output3 = row3.split('3')
print(output3)
```

Exercise

Write the function *capitalise_words* that:

- takes in and reads an input file consisting of words separated by commas and lines;
- capitalises all the letters and store the words in a list;
- display the list content.

In [ ]:
```
def capitalise_words(filename):
    f1 = open(filename, "r")
    result = []

    for line in f1:
        lst = line.strip().split(",")   # easiest to do using strip() to remo
        for i in range(len(lst)):
            lst[i] = lst[i].upper()
        result.extend(lst)

    print(result)

    f1.close()

capitalise_words("sample6.txt")
```

.strip() Method

Notice that we had to deal with the new line character ('\n') in the previous example.

There is an easy way to strip that out, as well as leading and trailing white spaces, using the **.strip()** method of a string when no argument is passed into it as shown in the examples below.

In [ ]:
```
word1 = "      hello     "
print(word1.strip())

word2 = "   he   llo    "
print(word2.strip())

word3 = "   he  llo \t hello \n \n \n"
print(word3.strip())
```

It is also possible to specify what you want to strip off instead.

In [ ]:
```
word4 = ",,,hello,"
print(word4.strip(','))

word5 = "hey hey hey hhh"
print(word5.strip('h'))
```

WITH Statement

In Python, the **with** statement is used when working with an external resource, such as a text file, a database file, etc. It ensures that the file is automatically closed when the nested code finishes running or when there is an exception that may otherwise potentially jeopardise the integrity of the data stored.

Take a look at the program code below and see how it differs from how we read a file earlier.

In [ ]:
```
lst = []

with open("sample7.txt") as f1:
    for line in f1:
        lst.append(line.strip())

print(lst)
```

CSV Module

The so-called **CSV (Comma Separated Values)** is the most common import and export format for spreadsheets and databases. Python's CSV module offers another way for us to read from and write into files.

Take a look at how the **.reader()** method behaves, paying close attention to the second argument supplied into the method.

In [ ]:
```
import csv

with open("sample8.txt") as f1:
    csv_file = csv.reader(f1, delimiter=',')
    for line in csv_file:
        print(line)
```

The following code shows how the **.writer()** and **.writerow()** methods behave. Take a look at the file that is created after you run the code.

In [ ]:
```
import csv

lst = ['eagle', 'fox', 'giraffe', 'horse']

with open("sample9.txt", "w") as f1:
```

```
    csv_file = csv.writer(f1, delimiter=',')
    csv_file.writerow(lst)
```

2020 JC1 H2 Computing 9569

11. Recursion

Recursion is a method of breaking down a problem into smaller sub-problems until the sub-problem is so small that it can be easily solved. It involves a function calling upon itself.

All recursive algorithms must obey two important rules:

- They must have a base case where the problem can be easily solved.
- They must change the state and move towards the base case.

A simple recursive function typically has the following construct.

In [ ]:
```
def recursive_function(n):
    if <problem is easy>:
        <do or return something>
    else:
        <do something>
        return recursive_function(<change n>)
```

To best understand the concept, we shall go through some examples.

3... 2... 1... Happy New Year!

Let us do a simple task of printing three integers in a descending order from 3 to 1, followed by a string "Happy New Year!"

Using a WHILE loop, write a code for the task described above.

In [ ]:
```
# Type your code here
n = 3
while n > 0:
    print(n)
    n -= 1
print("Happy New Year!")
```

If we do not want to use a FOR or a WHILE loop, can we still perform the task?

Study the code below and compare with your code above.

In [ ]:
```
def countdown(n):
    if n == 0:
        print("Happy New Year!")
    else:
        print(n)
        return countdown(n-1)

countdown(3)
```

The trace table below shows what happens at every step.

Step	n	Executed	Return
countdown(3)	3	print(3)	countdown(2)
countdown(2)	2	print(2)	countdown(1)

countdown(1)	1	print(1)	countdown(0)
countdown(0)	0	print("Happy New Year!")	None

In this example, the base case is when *n* = 0, where we print the string "Happy New Year" and stop calling the function recursively.

At each recursive call before *n* = 0, *n* must be decremented by 1 and passed as an argument when the same function is being called.

Recursive Sum

Consider the following recursive mathematical function.

---

f(x) = f(x-1) + 5, where f(0) = 2

Solve for f(5).

Solving by hand,

f(5) = f(4) + 5
f(4) = f(3) + 5
f(3) = f(2) + 5
f(2) = f(1) + 5
f(1) = f(0) + 5

Since f(0) = 2,

f(1) = 2 + 5 = 7
f(2) = 7 + 5 = 12
f(3) = 12 + 5 = 17
f(4) = 17 + 5 = 22
f(5) = 22 + 5 = 27

Notice that we have to evaluate f(5).

However, according to the definition, f(5) is in terms of f(4), so we need to evaluate f(4).

f(4) is in terms of f(3), so we need to evaluate f(3), which is in terms of f(2), which is in terms of f(1), which is in terms of f(0).

Every step leads us to another sub-problem that is closer to f(0).

Once we get to f(0) = 2, we are able to evaluate f(1), then f(2), then f(3), then f(4), then f(5).

Here is the recursive code to solve the problem.

In [ ]:
```
def f(n):
    if n == 0:
        return 2
    else:
        return f(n-1) + 5

print(f(5))
```

Product of Integers in a List

How can we find the product of integers in a list using recursion?

Suppose we have the following list: [1, 3, 5, 7].

We shall approach the problem by multiplying a pair of numbers at a time. The parentheses indicate the order we are going to perform the multiplications.

(((1 x 3) x 5) x 7)

Stepwise, the product of the integers can be calculated as such.

```
product = (((1 x 3) x 5) x 7)
        = ((3 x 5) x 7)
        = (15 x 7)
        = 105
```

Study the recursive code below to see how it works.

In [ ]:
```
def product(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return product(lst[:-1]) * lst[-1]

numbers = [1, 3, 5, 7]
print(product(numbers))
```

We can also parenthesise the expression the other way around.

(1 x (3 x (5 x 7)))

Stepwise, the product of the integers can be calculated as such.

```
products = (1 x (3 x (5 x 7)))
         = (1 x (3 x 35))
         = (1 x 105)
         = 105
```

Try to write the recursive code.

In [ ]:
```
def new_product(lst):
    # Type your code here
    if len(lst) == 1:
        return lst[0]
    else:
        return lst[0] * new_product(lst[1:])

numbers = [1, 3, 5, 7]
print(new_product(numbers))
```

Counting Even Numbers in a List

How do we count the number of even numbers in a list using recursion?

In [ ]:
```
def count_even(lst):
```

```
    if len(lst) == 0:
        return 0
    elif (lst[0] % 2 == 0):
        return count_even(lst[1:]) + 1
    else:
        return count_even(lst[1:])

print(count_even([1,2,3,4,5]))
```

Compared to the previous example on multiplication, why must the base case here be:

```
len(lst) == 0
```

instead of 1?

Exercises

Write the recursive code *recursive_sum* that takes in a list of integers and returns the sum.

In [ ]:
```
lst = [5, 6, 7, 8]

# Type your code here
def recursive_sum(lst):
    return lst[0] + recursive_sum(lst[1:]) if lst else 0

recursive_sum(lst)
```

Write the recursive code *count_capital* that takes in a string and returns the number of capital letters.

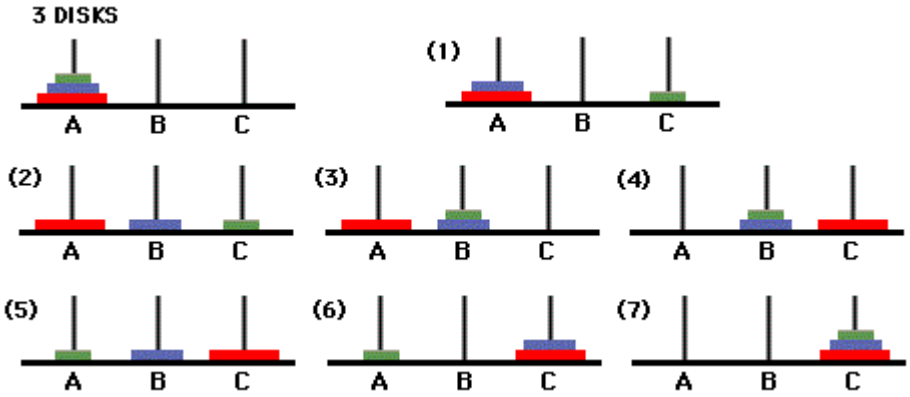
In [ ]:
```
str1 = "PuRple DinOsAUr"

# Type your code here
def count_capital(str1):
    return int(str1.isupper()) if len(str1) == 1 else int(str1[0].isupper()) +

count_capital(str1)
```

Write the recursive code *tuplify* that takes in a string and returns a tuple containing all the characters of the string.

In [ ]:
```
str2 = "aBCdE"

# Type your code here
def tuplify(str2):
    return (str2, ) if len(str2) == 1 else (str2[0], ) + tuplify(str2[1:])

tuplify(str2)
```

Tower of Hanoi

Have you heard of or played this puzzle before?



The Tower of Hanoi consists of three rods and a number of circular disks of different sizes. The puzzle begins with all the disks arranged on one of the rods (e.g. rod A) in order of size - largest at the bottom, smallest at the top.

The objective is to transfer all the disks from the initial rod to another under the following rules:

- Each move consists of moving a disk from one rod to another.
- You can only move one disk at a time.
- You cannot put a larger disk on top of a smaller one.

For 3 disks, it turns out that this can be done within 7 moves, as shown in the diagram above. (Is it possible to do solve it with fewer than 7 moves?)

Experiment a bit with solving the puzzle here:
https://www.mathsisfun.com/games/towerofhanoi.html

It is possible to solve it using iteration, although it may be complicated. A solution using recursion, on the other hand, is short and elegant.

By playing with the puzzle at the website above, try to come up with a method of moving the disks so that you can always do it in the shortest number of moves.

You will implement this in one of the missions in Coursemology.

2020 JC1 H2 Computing 9569

12. Object-Oriented Programming

In the early days, computers operated along a linear line of control, i.e. one could only do one thing at a time via Command-Line Interface (CLI). With the emergence of Graphical User Interface (GUI), this is no longer the case. You can type your report on a word processor, switching over to a web browser to search for something while also listening to songs from a music player at the same time. Under this multi-tasking environment, there is no longer a single thread of control in using computers. Not only can people do things in different orders, they can seemingly do many things concurrently. This is the motivation for **Object-Oriented Programming (OOP)**.

Many programs that we use today, such as Google (search engine), Facebook (social network) and Twitter (microblog) are developed using OOP.

Class, Object and Encapsulation

Imagine a situation where you need to write a program code to control the current balance of a bank account.

As a quick exercise, define the functions *deposit* and *withdraw*.

In [ ]:
```
def deposit(lst, money):
    lst[2] += money

def withdraw(lst, money):
    if lst[2] < money:
        return "Not enough money!"
    else:
        lst[2] -= money
```

In [ ]:
```
bank_account = ["912-83746-5", "Sam Phua", 20000]

deposit(bank_account, 1000)
print(bank_account)

withdraw(bank_account, 500)
print(bank_account)
```

What are some of the issues that may arise with such a simple program code?

In OOP, a **class** is a blueprint that defines the properties and methods of a group of similar objects. An **object** is a specific instance of a class that have the same properties and methods as the class from which it is built.

A class contains two components:

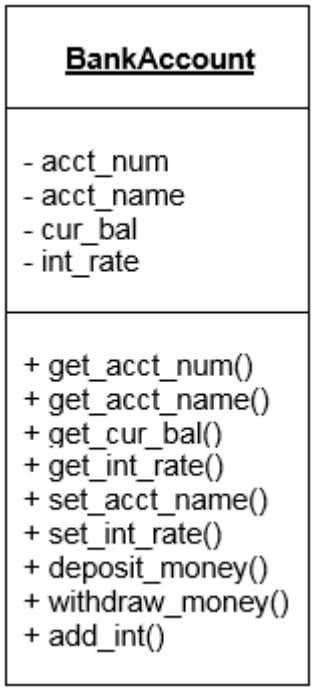
1. **Properties**: defining features of a class in terms of data
2. **Methods**: codes designed to perform particular tasks on the data

Encapsulation refers to the concept of bundling the properties and the methods together as a package. A recommended practice is to protect the data contained in the properties from being accidentally or intentionally modified by unauthorised parties. As such, we want to keep our data **private**. However, since the user needs a way to access them, we need to provide a set of methods to be made **public**. Take note that methods can be made private, but we shall not discuss about it here.

Defining a Class

In a bank account, we need to keep data such as account number, account holder name, current balance and annual interest rate. The typical operations done on a bank account include getting and setting the information of the account, deposit money, withdraw money and add interest.

A **Unified Modelling Language (UML)** diagram is typically used to illustrate OOP concepts. A table represents a class with the private data indicated by the **-** sign in front of the properties, while the public operations are indicated by having + in front of the methods as shown below.



The typical construct of a class is as follows.

In [ ]:
```
class <name>(<optional parent class>):

    def __init__(self, <optional parameters>):
        <constructor body>

    def <method name>(self, <optional parameters>):
        <method body>

    ...
```

Methods of a class can generally be classified into a few types:

1. **Constructor**

   The **__init__** function allocates storage when an object of a class is created.

1. **Accessor**

   The 'get' functions access the data stored in an object.

1. **Mutator**

   The 'set' functions allow modification to the data stored in an object.

1. **Utility**

   These methods extend the functionality of the class, e.g. deposit money in our *BankAccount* class.

Notice that the construct contains **self**, which is used to refer to itself and is required in each of the methods of a class.

Let us define our *BankAccount* class.

In [ ]:
```
class BankAccount:
    # Constructor
    def __init__(self, acct_num, acct_name, cur_bal, int_rate):
        self.acct_num = acct_num
        self.acct_name = acct_name
        self.cur_bal = cur_bal
        self.int_rate = int_rate

    # Accessors
    def get_acct_num(self):
        return self.acct_num

    def get_acct_name(self):
        return self.acct_name

    def get_cur_bal(self):
        return self.cur_bal

    def get_int_rate(self):
        return self.int_rate

    # Mutators
    def set_acct_name(self, name):
        self.acct_name = name

    def set_int_rate(self, int_rate):
        self.int_rate = int_rate

    # Utllity methods
    def deposit_money(self, money):
        self.cur_bal += money

    def withdraw_money(self, money):
        if money > self.cur_bal:
            return "Not enough money!"
        else:
            self.cur_bal -= money

    def add_int(self):
        self.cur_bal += self.cur_bal * self.int_rate
```

Creating and Manipulating an Object

The syntax to create an object of a class is as follows.

In [ ]:
```
<object_name> = <class_name>(<required parameters defined in __init__>)
```

We can now create an object of the *BankAccount* class. Let us call it *david_account* with the following data to be supplied:

- Account number: 123-45678-9
- Account name: David Tan
- Current balance: 100000
- Annual interest rate: 0.1%

In [ ]:
```
david_account = BankAccount("123-45678-9", "David Tan", 100000, 0.001)
```

Once the object has been created, we shall test the correctness of the methods defined earlier.

In [ ]:
```
# Test accessor methods

print(david_account.get_acct_num())
print(david_account.get_acct_name())
print(david_account.get_cur_bal())
print(david_account.get_int_rate())
```

In [ ]:
```
# Test mutator methods

david_account.set_acct_name("David Tan Ming Quan")
print(david_account.get_acct_name())

david_account.set_int_rate(0.002)
print(david_account.get_int_rate())
```

In [ ]:
```
# Test utility methods

# This should print 110000
david_account.deposit_money(10000)
print(david_account.get_cur_bal())

# This should print 90000
david_account.withdraw_money(20000)
print(david_account.get_cur_bal())

# This should print "Not enough money!"
david_account.withdraw_money(200000)

# This should print 90180.0
david_account.add_int()
print(david_account.get_cur_bal())
```

After going through the example above, you should realise that it is actually not something entirely alien to you. You have, perhaps unknowingly, been using OOP concepts when you write codes involving strings, lists and tuples, but to name some that we have previously covered.

In [ ]:
```
# Instantiate an object of the list class
my_list = list()

# Perform some methods on the list object
```

```
my_list.append('a')
my_list.append('b')
print(my_list)

my_list.pop()
print(my_list)
```

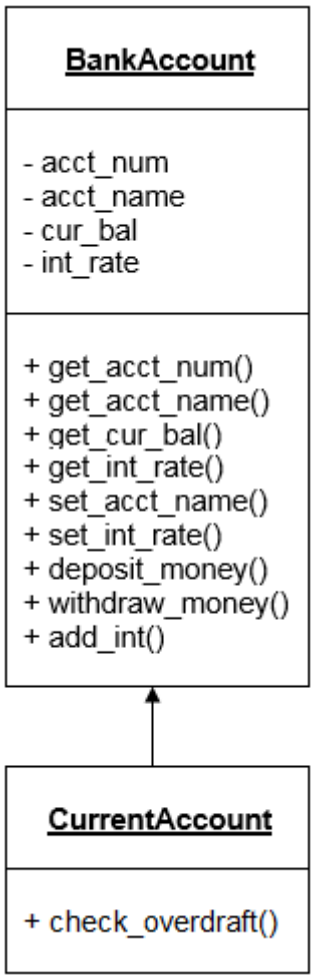
Inheritance

Inheritance refers to the concept of properties and methods in one class being shared with its subclass. This promotes code reusability.

The **subclass** inherits all the properties and methods of the **superclass**, but the former may behave differently from the latter with the addition or modification of certain features.

To illustrate this concept, we shall look at a *CurrentAccount*, a subclass of *BankAccount*. A current account typically caters for frequent deposits and withdrawals by cheque. Something unique to this type of account is that it may go into an overdraft, a state where the available balance goes below zero. Also, it is often regarded as a non-interest bearing account, or the interest rate is usually very small, e.g. 0.001%.

Here is the UML diagram to show the relationship between *BankAccount* and *CurrentAccount*.



The typical construct of a subclass is as follows.

In [ ]:
```
class <name>(<superclass>):
```

```
    def <additional / overriding method name>(self, <optional parameters>):
        <method body>

    ...
```

To define the *CurrentAccount* subclass, we simply need to add the additional method *check_overdraft* that should return True if the current balance is below zero, and False otherwise.

In [ ]:
```
class CurrentAccount(BankAccount):

    def check_overdraft(self):
        if self.cur_bal < 0:
            return True
        else:
            return False
```

Let us create the object *ali_account* of the *CurrentAccount* class with the following data to be supplied:

- Account number: 123-45678-9
- Account name: Ali Ramlan
- Current balance: 100000
- Annual interest rate: 0.1%

In [ ]:
```
ali_account = CurrentAccount("123-45678-9", "Ali Ramlan", 100000, 0.001)
```

Run the following codes to see for yourself how an object of *CurrentAccount* behaves.

In [ ]:
```
# Test accessor methods

print(ali_account.get_acct_num())
print(ali_account.get_acct_name())
print(ali_account.get_cur_bal())
print(ali_account.get_int_rate())
```

In [ ]:
```
# Test mutator methods

ali_account.set_acct_name("Ali Ramlan")
print(ali_account.get_acct_name())

ali_account.set_int_rate(0.00002)
print(ali_account.get_int_rate())
```

In [ ]:
```
# Test utility methods

# This should print 12000
ali_account.deposit_money(2000)
print(ali_account.get_cur_bal())

# This should print 11500
ali_account.withdraw_money(500)
print(ali_account.get_cur_bal())

# This should print False
print(ali_account.check_overdraft())
```

Polymorphism

Polymorphism refers to the concept of a subclass method, which is inherited from its

superclass, used in a different way. The literal meaning of polymorphism is to take on many shapes.

Polymorphism is realised through **method overriding**, which refers to the redefining of the implementation of a method provided by the superclass. This allows for generalisation of method name, which may behave slightly differently depending on the subclasses we are dealing with.

For our *CurrentAccount* subclass, we need to redefine the following utility methods:

```
+ withdraw_money()   allows cur_bal to go below zero
+ add_int()          if overdraft, 5% should be applied to the
debt annually
                     otherwise, it should do the same as the
original superclass method
```

In [ ]:
```python
class CurrentAccount(BankAccount):

    def check_overdraft(self):
        if (self.cur_bal < 0):
            return True
        else:
            return False

    # Redefine the two utility methods
    def withdraw_money(self, money):
        self.cur_bal -= money

    def add_int(self):
        if self.check_overdraft():
            self.cur_bal += self.cur_bal * 5/100
        else:
            self.cur_bal += self.cur_bal * self.int_rate
```

Run the following codes to test the correctness of your utility methods.

In [ ]:
```python
kannan_account = CurrentAccount("876-54321-0", "Kannan Kumar", 800, 0.00001)

# The account goes to an overdraft after withdrawing $1,800 to become -$1,000
kannan_account.withdraw_money(1800)
print(kannan_account.get_cur_bal())

# An interest of 5% is applied to the overdraft account to become -$1,050
kannan_account.add_int()
print(kannan_account.get_cur_bal())

# $2,000 is deposited into the account to become $950, so the account is not
kannan_account.deposit_money(2000)
print(kannan_account.get_cur_bal())

# The default interest rate is applied to the non-overdraft account to become
kannan_account.add_int()
print(kannan_account.get_cur_bal())
```

Built-in Libraries

These are some useful libraries that we need to know how to use, which need to be **imported** first before we write our program codes.

1. Random Library

Method	Return
random()	A random float in the range [0.0, 1.0), i.e. between 0.0 (inclusive) and 1.0 (exclusive)
randint(a, b)	A random integer in the range [**a**, **b**], i.e. between **a** (inclusive) and **b** (inclusive)
randrange(stop)	A random integer in the range [0, **stop**), i.e. between 0 (inclusive) and **stop** (exclusive)
randrange(start, stop)	A random integer in the range [**start**, **stop**), i.e. between **start** (inclusive) and **stop** (exclusive)
randrange(start, stop, step)	A random integer in the range [**start**, **stop**), i.e. between **start** (inclusive) and **stop** (exclusive), in intervals of **step**
shuffle(lst)	Shuffles the elements in list **lst** without any return value

Try running the codes below to see the outputs.

In [ ]:
```
import random

print(random.random())
print(random.randint(0, 5))
print(random.randrange(5))
print(random.randrange(5, 10))
print(random.randrange(1, 10, 2))

lst = [2, 0, 1, 9]
random.shuffle(lst)
print(lst)
```

2. Math Library

Try to work out the output of each of the following before running the code. You need to use a calculator to evaluate some of them.

In [ ]:
```
import math

print(math.trunc(20.19))
# Output: 20

print(math.floor(20.19))
# Output: 20

print(math.ceil(20.19))
# Output: 21

print(math.pow(2, 3))   # VS 2**3
# Output: 8.0 (float)   # VS 8 (integer)

print(math.exp(2))
# Output: 7.38905609893065

print(math.log(10))
# Output: 2.302585092994046

print(math.sqrt(4))
# Output: 2.0
```

3. Datetime Library

Note that this library uses the 24-hour clock convention.

Method	Return
trunc(n)	An integer after removing any decimal values of **n**
floor(n)	An integer after rounding **n** down
ceil(n)	An integer after rounding **n** up
pow(n, x)	A float when **n** is raised to the power of **x**
exp(n)	A float when *e* (~2.7) is raised to the power of **n**
log(n)	A float that is the logarithm to the base of *e* (~2.7) of **n**
sqrt(n)	A float that is the square root of **n**

Method	Return
datetime.now()	A *datetime* object representing the current date and time
datetime(year, month, day, {hour, {minute, {second, {microsecond}}}})	A *datetime* object representing the specified date and time
datetime.strptime(str, format)	A *datetime* object from a given string **str** of a given **format**
< datetime >.strftime(format)	A string of a given **format** representing the *datetime* object
< datetime >.isoformat()	A string representing the date and time in ISO 8601 format (YYYY-MM-DDTHH:MM:SS)

Try running the codes below to see the output.

In [ ]:
```
import datetime

datetime.datetime.now()
```

In [ ]:
```
test_date1 = datetime.datetime(2019, 1, 1)

print(test_date1.isoformat())
```

In [ ]:
```
str_date = "01/04/19 13:55:26"
str_format = "%d/%m/%y %H:%M:%S"

test_date2 = datetime.datetime.strptime(str_date, str_format)

print(test_date2)
```

In [ ]:
```
test_date3 = datetime.datetime(2019, 4, 1, 8, 20, 33)
new_format = "Date: %d-%m-%y \nTime: %H:%M:%S"

print(test_date3.strftime(new_format))
```

To get the specific component of the object, the data of the datetime library are officially made accessible for public use.

Data	Return
datetime.now().year	An integer representing the current year
datetime.now().month	An integer representing the current month
datetime.now().day	An integer representing the current day
datetime.now().hour	An integer representing the current hour
datetime.now().minute	An integer representing the current minute
datetime.now().second	An integer representing the current second

Try running the codes below to see the output.

In [ ]:
```
test_date4 = datetime.datetime(2019, 12, 25, 15, 30, 45)

print(test_date4.year)
print(test_date4.month)
print(test_date4.day)
print(test_date4.hour)
print(test_date4.minute)
print(test_date4.second)
```

It is also possible to find the difference between two *datetime* objects in terms of days and seconds respectively.

Method	Return
< timedelta >.days	An integer representing the difference in terms of number of days
< timedelta >.seconds	An integer representing the difference in terms of number of seconds

Try running the codes below to see the output.

In [ ]:
```
d1 = datetime.datetime(2019, 1, 1, 21, 10, 10)
d2 = datetime.datetime(2020, 1, 1, 21, 10, 10)

days_to_go = (d2-d1).days

print(days_to_go)
```

In [ ]:
```
d3 = datetime.datetime(2019, 12, 25, 8, 0, 0)
d4 = datetime.datetime(2023, 12, 25, 9, 1, 5)

# Note that this considers only the difference in time and disregards the dat
secs_to_go = (d4-d3).seconds

print(secs_to_go)
```

In [ ]:
```
d5 = datetime.datetime(2020, 1, 1)
diff = datetime.timedelta(days=100)

# What is 100 days later after 1 January 2020?
d6 = d5 + diff
print(d6)
```

2020 JC1 H2 Computing 9569

13. Storing Characters and Numbers

Storing Characters ¶

How are characters stored in computers?

Any group of 0s and 1s can be used to represent a specific character. The number of bits used to store one character is called a **byte**, which usually comprises eight bits. The complete set of characters that a particular computer uses is known as its **character set**.

Consider a three-bit system to represent upper case letters in alphabetical order as shown in the table below.

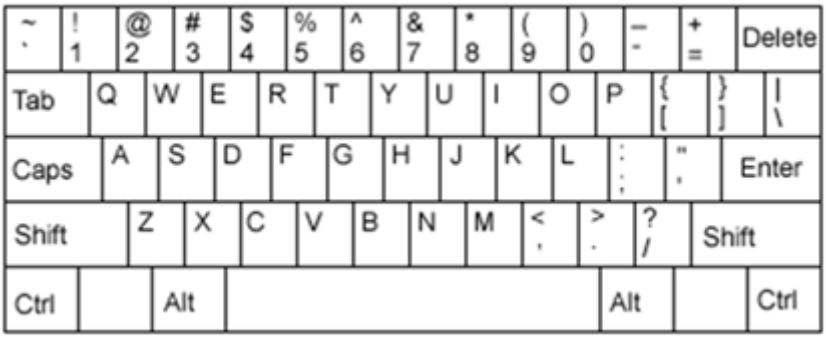
Sequence of Bits	Corresponding Letters
000	'A'
001	'B'
010	'C'
011	'D'
100	'E'
101	'F'
110	'G'
111	'H'

It can be seen that using a three-bit system only allows for eight possible unique codes. That is to say, we can only store the upper case letters 'A' to 'H', but not the rest of the alphabet, not to mention their lower case counterparts, punctuation marks, decimal digits and so on.

Some systems do not need to be able to recognise a lot of characters, so a few bits for each character is sufficient. One example is an Automated Teller Machine (ATM) that we use to perform financial transactions.

As a quick thinking exercise, how many bits are necessary to encode the required number of characters used in an ATM? Assume that we only have the digits 0 to 9, as well as three operations: enter, delete and cancel?

Let us look at the typical layout of a QWERTY keyboard, which gives a quick glance into a portion of the character set of the computers that most of us use in our daily lives.



Imagine a situation where one computer uses 10000001 to represent the upper case letter ‘A’, while another computer represents the same letter with 10000010. Any document files created on one of the computers is not going to make sense to the other as the two will interpret the codes differently. As such, standardisation is required in order for computers to be able to communicate with each other.

American Standard Code for Information Interchange (ASCII)

In October 1960, the work on the **American Standard Code for Information Interchange (ASCII)** began and it became the first character encoding system to be used across the globe. Each ASCII character is represented by a sequence of bits or a byte.

The original ASCII system uses the decimal numbers 0 to 127 (i.e., the binary numbers 0000000 to 1111111) to encode each character, so each character is represented by seven bits.

(An eighth bit was used as a **check digit** if the computer manufacturer wished to do so. Some computer manufacturers simply set the eighth bit to 0 for all characters. Because of this, the convention that a byte represented eight bits began. Check digits will be explained in the chapter on Debugging.)

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Another version of the table can be found at http://www.asciitable.com/ (http://www.asciitable.com/)

- Characters 0 to 31 (0000000 to 0011111), as well as 127 (1111111) are known as **control characters**. These were previously used to assist in data transmission or entering data at a computer terminal, as well as controlling the output when a computer printed out its output directly on paper without displaying it on a screen. They have very limited use in modern computing.

- Characters 48 to 57 (0110000 to 0111001) represent the digits 0 to 9. The choice of this range provides an easy way for a human to recognise a denary digit in ASCII - if it starts with 011, the person can (mentally) convert the remaining 4 binary digits into a denary number.

- Characters 65 to 90 (1000001 to 1011010) represent the uppercase letters A to Z, and characters 97 to 122 (1100001 to 1111010) represent the lowercase letters a to z. Again, this provides an easy way for a human to recognise a letter of the alphabet. If it starts with 10 or 11, it is a uppercase or lowercase letter respectively and the remaining 5 bits indicate the position of the letter in the alphabet (e.g. M is 13 which is 1101 in binary, so uppercase M is 1001101 in ASCII.)

- The remaining characters (32 to 47, 58 to 64, 91 to 96, and 123 to 126) represent various punctuation symbols.

The ASCII encoding of a character can be found using the Python function `ord`. Likewise, a denary number can be converted into its corresponding character using the Python function `chr`.

In [ ]:

```
print(ord('A'))
print(ord('a'))
print(ord('@'))
print(chr(65))
print(chr(97))
print(chr(64))
```

The ASCII encoding system is adequate for the English language and some others that use a similar alphabet, such as Latin and Malay. In time, speakers of other languages would develop encoding systems for their own languages.

IBM released an extension of ASCII called 'Code page 437' which would eventually become known as Extended ASCII. These included additional punctuation symbols, mathematical symbols, and some characters used in European languages (such as Æ,ñ, and ö).

The Extended ASCII table is shown below.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	▒	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▓	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	Ö	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	▉	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	Pts	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

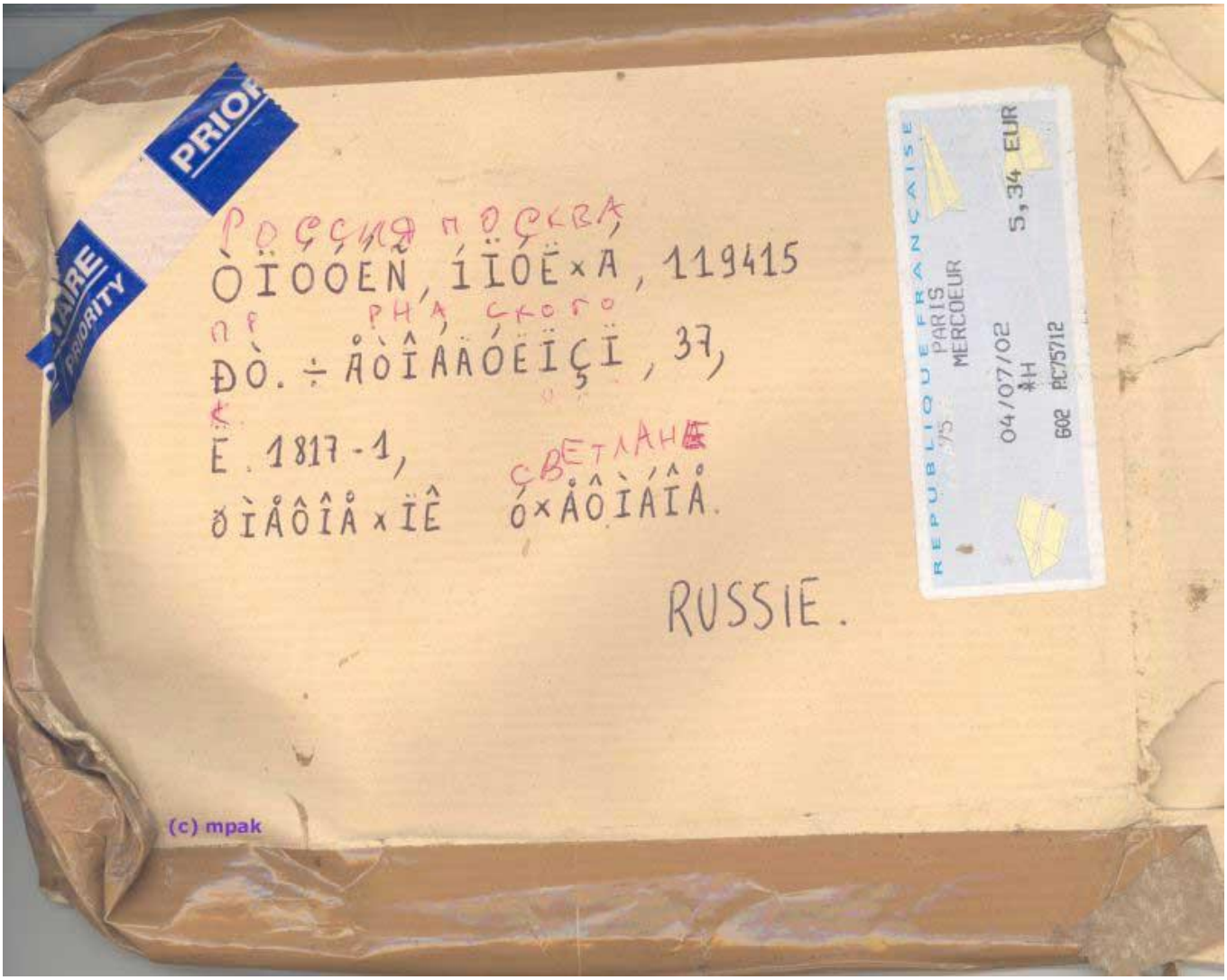
Since the numbers now run from 0 to 255, eight bits (one byte) are now required to encode all the characters in the Extended ASCII table without using any check digits. The original ASCII characters would have a 0 added to the front. Since four binary digits (bits) correspond to one hexadecimal digit, this extended ASCII system uses two hexadecimal digits to represent each character in the system.

Some countries, such as India and Vietnam, would eventually create their own extensions of ASCII (known as ISCII and VISCII) respectively. These are extensions in the sense that 0 to 127 would still correspond to the original ASCII characters, and the additional characters used for other languages would be encoded with numbers greater than 127. This allows for **compatibility** with ASCII users, since a text encoded with ASCII would still be able to be read normally in the other encoding systems without any modification.

Establishing a Global Standard

Unfortunately, there are some languages that are used in multiple countries or regions, each of which developed their own encoding systems that may have been compatible with ASCII but not with each other. For instance, Chinese was encoded primarily using the GB (Guobiao) system in China and Singapore, and the Big5 system in Taiwan, Hong Kong and Macau. This meant that transmissions from one country to another could turn out garbled.

The following parcel, for instance, was sent from a French person to her Russian friend.



The Russian friend emailed the French person her address in Russia, but as their two computers used different encoding systems, the address, which was meant to be written in the Russian alphabet, was decoded using a different system. Fortunately, the Russian post office realised the problem and managed to find the corresponding Russian characters, and delivered the parcel correctly.

In an effort to avoid such problems, the computing industry tried to establish a global standard. There were two ways it could have gone about this:

What does not work: assigning each character more bytes

One simple way to establish a global standard would be to determine at the outset how many characters would be encoded into the standard, and work out how many bytes would be needed for each character. For instance, assuming 2^{24} = 16,777,216 characters are needed to encode the whole world's languages, then we would only need 24 bits, or 3 bytes, to represent every possible character. This sounds like a reasonable assumption, but it does have disadvantages, namely:

- This system is not backwards compatible with ASCII. This means that all previous files and programs which were stored using ASCII would need to be converted into the new system, or a program for converting ASCII files into the new system would need to be written.
- Each character in this system would be 3 bytes, but the vast majority of files and programs in existence are already written using ASCII, with only 1 byte per character. All these files and programs would need to triple in size to accommodate the new system.
- The system is not extendible. As technology spreads around the world, there is a need for an increasing number of languages, with their written forms, to be computerised. In the (admittedly unlikely) event that the predetermined number of bytes per character is not enough, the computing industry would once again need to determine a new standard, and it would still not be backwards compatible with either ASCII or the older standard.

While some of the national standards, especially for countries such as China, Japan and Korea, used this system to extend ASCII to encode their languages, ultimately, it was decided that this would not be an appropriate way to encode a global standard which would need many more characters.

Therefore, a system called Unicode was drawn up which would overcome the drawbacks mentioned above.

Unicode

The first volume of this standard was published in October 1991. The ultimate aim of Unicode is to be able to present any possible text in any written language, in code form. This has been extended to include a number of other symbols used in technical situations, as well as emoji. Unicode is designed so that once a code has been determined, it never changes.

Unicode has its own special terminology. A character code is referred to as a **code point**. Currently, there are three standards, known as UTF-8, UTF-16 and UTF-32 (UTF stands for ‘Unicode Transformation Format’). The main difference between Unicode and the system described above is that **Unicode encodes each character with a different number of bytes**. The system will be described in the next section (not in syllabus).

Python 3.6 is compatible with UTF-8. UTF-8 encodes all characters using one to four eight-bit bytes. The one-byte characters are the original ASCII characters. **This ensures that old files and programs written in ASCII can still be read in Unicode without any conversion required, and, more importantly, their file size is still the same.**

The compatibility of UTF-8 with the original (non-Extended) ASCII encoding is demonstrated by the table below. Note the Unicode terminology - the number representing the character is known as a **code point**, and is written with 'U+' followed by a 4 (or more)-digit hexadecimal number. This system makes it easy for people to refer to specific characters within the Unicode system.

Character	ASCII (denary)	ASCII (7-bit binary)	ASCII (hexadecimal)	UTF-8 code point
$	36	0100100	24	U+0024

Other characters, not in the original ASCII set, also have their own Unicode encodings.

Character	UTF-8
¢	U+00A2
ह	U+0939
€	U+20AC
𐍈	U+10348
😹	U+1F431

The Python functions `ord` and `chr` actually contain the Unicode characters and not just the Python ones (although they may not print correctly if your computer does not have the appropriate fonts installed.)

In Python, the code points are represented using the control sequence `"\u"` followed by the hexadecimal numbers.

In [ ]:

```
print('\u03C0')
print(ord('\u03C0'))
print(chr(960))
```

Unicode in detail (not in syllabus)

The Unicode system can be seen in more detail in the following table (**not in syllabus**):

Number of bytes	Byte 1	Byte 2	Byte 3	Byte 4	Number of bits for encoding	Number of possible characters	First code point	Last code point
1	0xxxxxxx				7	2^7=128	U+0000	U+007F
2	110xxxxx	10xxxxxx			11	2^{11}=2,048	U+0080	U+07FF
3	1110xxxx	10xxxxxx	10xxxxxx		16	2^{16}=65,536	U+0800	U+FFFF
4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	21		U+10000	U+10FFFF

(In practice, for reasons not related to the encoding system, not all 2^{21} possible combinations are used for 4-byte encodings. As a result, "only" 1,112,064 characters are allowed in UTF-8.)

Therefore, as a computer reads each byte, it knows how many bytes to expect for the current character. If the byte it is reading starts with '0', then it is a one-byte character. If the byte it is reading starts with '1110', then it is a three-byte character, so this byte and the next two comprise one single character.

There is also some degree of error checking involved. For instance, if one byte starts with '1110' and the following two bytes do **not** both start with '10', then there is an error somewhere and the file has been corrupted.

The following examples show how characters are encoded in UTF-8:

Character	Code point	Number of bytes	Conversion to binary	Unicode
¢	U+00A2	2	0000 0000 1010 0010	11000010 10100010
ह	U+0939	3	0000 1001 0011 1001	11100000 10100100 10111001
€	U+20AC	3	0010 0000 1010 1100	11100010 10000010 10101100
𐍈	U+10348	4	0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000
😹	U+1F431	4	0001 1111 0100 0011 0001	11110000 10011111 10010000 10110001

Storing Numbers

Strictly, this section is **not in syllabus**. However, it is **extremely useful** to know why your computer programs sometimes have inexplicable rounding-off errors, and what you can do to avoid them.

Integers

Integers are stored in binary notation. For instance, a computer program may dedicate one byte (eight bits) to storing an integer. This means that the integer can be between 00000000 (0 in denary) and 11111111 (255 in denary). The main advantage of the binary system, apart from the fact that it fits naturally with the electronic components used to make computers, is that very few arithmetic operations need to be hard-coded into the computer for it to be able to carry out mathematical operations.

For example, binary addition only has four rules:

- 0 + 0 = 0
- 1 + 0 = 1
- 0 + 1 = 1
- 1 + 1 = 0 and carry 1.

Likewise, binary multiplication only has four rules as well:

- 0 × 0 = 0
- 1 × 0 = 0
- 0 × 1 = 0
- 1 × 1 = 1

Performing operations in binary works as long as the final answer is within the range of integers that can be encoded with one byte (0 to 255).

This system can be modified in a number of ways:

- If we want to store larger numbers, we would need more than one byte to do so. Most computer systems use either two or four bytes to store integers, allowing them to go up to 2^{16} - 1 = 65535 or 2^{32} - 1 = 4294967295 respectively. Nevertheless, if this is not pre-empted as the result of a calculation the computer is making, it can lead to an **overflow error**.
- We don't have a good way of storing negative numbers. This can be addressed by either of the following methods.

Use one bit for the sign

The first bit, which we also call the **most significant bit (MSB)** can represent the sign, for instance, 0 for positive numbers and 1 for negative numbers. The remaming seven bits are used to indicate the magnitude (absolute value) of the number. Therefore, using one byte, we can represent numbers between -127 and +127. However, we have to modify the arithmetic operations described above to allow for adding negative numbers. Furthermore, this system also has a positive and negative zero (10000000 and 00000000), which are two different encodings for the same number.

Two's complement

In the original binary system, each bit represents one of the powers of two, from 2^7 being the most significant bit to 2^0 being the least significant bit. As we read from left to right, we add up that power of 2 if the corresponding bit is 1, and ignore it otherwise.

In the Two's complement system, the most significant bit represents a *negative* number, -2^7. The remaining bits represent positive powers of 2 as usual. Therefore, if the most significant bit is 0, the remaining seven bits are the usual binary representation of a number from 0 to 127. If the most significant bit is 1, however, we decode the remaining seven bits as a number between 0 and 127, and *subtract* 128 from it.

For example, 01110101 ($2^6 + 2^5 + 2^4 + 2^2 + 2^0$) represents the number 117 . On the other hand, 10001011 ($-2^7 + 2^3 + 2^1 + 2^0$) represents the number -117. The advantage of this method is that the addition rules mentioned above continue to work as long as the result is in the range of numbers that can be encoded (-128 to 127).

Another way to obtain the encoding of a negative number is to write down the *magnitude* of the number in binary (8 bits), then change all the digits - changing 1 to 0, and vice versa (hence the 'complement' in the name), and then adding 1 to the final answer. For instance, +117 is encoded as 01110101 (we only take the last 7 bits). Taking the complement of the digits gives 1001010, and adding 1 gives 1001011.

- Finally, we do not have a good way to represent fractions or decimals in this form. The usual way to do this now is to use the floating point notation, described below.

Real numbers

In denary, the digits after the decimal point indicate negative powers of 10. For instance,

$23.456 = 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$.

We can write it in the form $a \times 10^b$ where $-1 < a < 1$. (This is similar to scientific notation, but the range of allowed values of a is from 0 to 1, instead of 1 to 10.) In this case,

$23.456 = 0.23456 \times 10^2$.

In such a notation, $a = 0.23456$ is called the **mantissa** and $b = 2$ is called the **exponent**.

Likewise, a number like 0.0000134 would be written as 0.134×10^{-4}. In this case, 0.134 is the mantissa and -4 is the exponent.

Likewise, in the binary system, the digits after the **bicimal point** (also called the **radix point**) indicate negative powers of 2. For instance,

$10.1011 = 2^1 + 2^{-1} + 2^{-3} + 2^{-4}$ (which is 2.6875 in denary).

However, we can write it as

$10.1011 = 0.101011 \times 2^2$,

so that 0.101011 is the mantissa and 2 is the exponent.

Similarly,

$0.0001101 = 0.1101 \times 2^{-3}$,

so that 0.1101 is the mantissa and -3 is the exponent.

We can thus store any real number as a mantissa and an exponent. This is known as a **floating point representation** because the bicimal or decimal point floats into position, depending on the value of the exponent.

Suppose we want to store a number using two bytes. We can use the first byte to store the mantissa and the second byte to store the exponent.

For example, the denary number 112 would be encoded in the following way:

$112_{10} = 1110000_2 = 0.1110000 \times 2^{111}$ (remember the exponent is also in binary).

Hence it would be encoded as 01110000 00000111.

The binary number 10.11011 = 0.1011011 x 2^{10} would be encoded as 01011011 00000010.

The binary number 0.00000101011 = 0.1010110 x 2^{-101} would be encoded as 01010110 11111011 (the negative exponent is encoded using Two's complement)

The negative binary number -1011 is encoded using the following steps.

1. -1011 = -0.1011 x 2^{100}.
2. The magnitude of the mantissa is 0.1011. We use the Two's complement method (changing 1s and 0s, and then adding 1 at the rightmost bicimal place) to get 1.0101.
3. Therefore, the mantissa would be encoded as 10101000.
4. The exponent is 00000100.

The negative binary number -0.01001 (which is 11/32 in denary) is encoded using the following steps.

1. -0.01001 = -0.1011 x 2^{-1}.
2. The magnitude of the mantissa is 0.1001. We use the Two's complement method (changing 1s and 0s, and then adding 1 at the rightmost bicimal place) to get 1.0111.
3. Therefore, the mantissa would be encoded as 10111000.
4. The exponent is a negative number, and hence would also be encoded using Two's complement to get

In [ ]:

```
print(0.1 + 0.2)
```

0.1 in binary is 0.0001100110011... 0.2 in binary is 0.0011001100110... These are stored as floating point numbers in the computer, but since they are infinitely recurring bicimal numbers, the mantissa has to be truncated at a certain number of significant figures before the mathematical operation can be carried out. This results in a loss of precision in the final answer.

For simplicity, assume that the calculation is carried out to 7 significant figures.

The computer thus adds 0.001100110 to 0.0001100110 to get 0.01001101. Translating this back into denary notation, this is

$2^{-2}+2^{-5}+2^{-6}+2^{-8} = 0.25 + 0.03125 + 0.015625 + 0.00390625 = 0.30078125$

As a result, if floating point numbers are stored up to 7 significant (binary) figures, we do not expect accuracy beyond about 3 significant figures in decimal.

This affects many functions which operate on floating point numbers.

In [ ]:

```
print(round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1))
```

In [ ]:

```
print(round(.1 + .1 + .1, 10) == round(.3, 10))
```

The above explanation is a simplified explanation. **Python floating point numbers are accurate to 16 significant figures in base 10**.

Problems with floating point numbers

Many uses of floating-point numbers are in extended mathematical procedures involving repeated calculations. Examples of such use would be in weather forecasting using a mathematical model of the atmosphere or in economic forecasting. In such programming, there is a slight approximation in recording the result of each calculation. These rounding errors can become significant if calculations are repeated enough times. The only way of preventing this becoming a serious problem is to increase the precision of the floating-point representation by using more bits for the mantissa. Some programming languages offer options two work in 'double precision' (or 'quadruple precision') where double (or quadruple) the usual number of bytes are used to store the mantissa.

Another problem is the range of numbers that can be stored. While the range of numbers is much larger than what can be stored as an integer, there is also a possibility that if a very small number is divided by a very large one, the result in a value smaller than the smallest possible number that can be stored. This is an **underflow** error. Some programming languages treat such a small number as zero; depending on the nature of the calculation involved, it may lead to errors later on.

In [ ]:

```
print(2**50)
print(2**50 + 0.1)
```

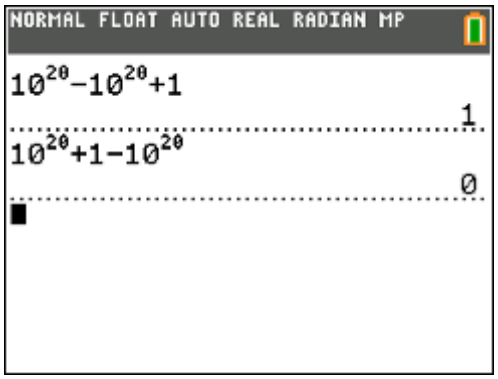
In [ ]:

```
print(10000+0.1-10000)
print(10000-10000+0.1)

#why do the two outputs differ?
```

In [ ]:

```
for i in range(10):
    print(1/(2**(1000+i*10)))

#notice the loss in the number of significant figures as we get to very small numbers.
```

Most calculating devices which have to deal with real numbers use some kind of floating point system. This includes your graphing calculator! Try some of the calculations on your GC and see what happens!



Appendix

For your watching pleasure!

Unicode and Character Encoding

- https://www.youtube.com/watch?v=wCQSIub_g7M (https://www.youtube.com/watch?v=wCQSIub_g7M)
- https://www.youtube.com/watch?v=MijmeoH9LT4 (https://www.youtube.com/watch?v=MijmeoH9LT4)
- https://www.youtube.com/watch?v=qBex3IDaUbU (https://www.youtube.com/watch?v=qBex3IDaUbU)
- https://www.youtube.com/watch?v=5OPkGQoPeHk (https://www.youtube.com/watch?v=5OPkGQoPeHk)

Floating Point Numbers

- https://www.youtube.com/watch?v=PZRI1IfStY0 (https://www.youtube.com/watch?v=PZRI1IfStY0)

2020 JC1 H2 Computing 9569

14. Debugging

Programmers make all kinds of mistakes when writing computer programs. These can be due to various reasons, including (but certainly not limited to):

- assumptions about the input
- misunderstanding about the algorithm
- incorrect calculations
- poorly designed data structures
- blunders

Testing your program is a way to find errors before it is released as a product. When done well, the testing would be able to tell us where the errors are.

Errors

In general, there are three kinds of errors:

- syntax errors
- runtime errors
- logic errors

Syntax errors are errors in the use of the programming language. Some examples include:

- forgetting a colon (`:`) at the end of a header such as a `def` or `if` statement
- misspelling a certain keywords (e.g. def, for, while)

Runtime errors are errors that arise when Python is asked to perform operations that cannot be done. Some examples include:

- misspelling of a previously declared variable
- dividing an integer by zero
- asking for `lst[10]` in a list `lst` that only has 10 elements inside

Sometimes the error may go undetected for some time. Referring to the second example, for instance, if the majority of the user input does not cause `lst[10]` to be called, the error may go undetected until another user keys in an input that calls for `lst[10]` .

Logic errors are a broad class of errors that encompasses everything else not covered in the previous two categories. While the syntax is correct and Python can execute the program without throwing any error messages, the results are not as intended. Some examples include:

- incorrectly written mathematical expression
- a condition that is left out in an `if - else` statement
- skipping a value in a list

Debugging

Errors are also known as **bugs**. Thus, the process of finding and correcting errors is called **debugging**.

In general, debugging comprises three steps:

1. **Test**: run tests to determine whether the program works as intended
2. **Probe**: determine where the error occur
3. **Fix**: fix the error

To test the program, we generally make use of **test cases**, which are inputs where we know what the expected outputs are supposed to be. We usually start with some simple ones to make sure that the program works as expected. Once that is done, we should take a more adversarial approach by trying out inputs that might cause errors, such as extreme values or those that might be exceptional in some way or another.

Once a problem has been discovered, we need to **probe** the program to locate its source(s). At first, we can try to **read** through the code, or **explain** it to someone what each line is supposed to do. As an aside, did you know that many professional programmers keep a small toy or doll on their desk and explain their code to it as part of the debugging process?

If that does not work, we can employ a technique called **code tracing** to visualise the flow of execution of the processes by inserting **print statements** as probes.

We shall go through some examples together.

(a) aim: to find the sum of integers from 1 to 100

In [ ]:
```
# NameError: number and the_sum have not been defined yet

while number = 100:   # logic error: should be <=
                      # also a syntax error because a single = cannot be used
    the_sum = the_sum + number

print("The answer is", the_sum)
```

In [ ]:
```
number = 1
the_sum = 0

# logic error: infinite loop since number is not incremented
while number <= 100:
    the_sum = the_sum + number

print("The answer is", the_sum)
```

In [ ]:
```
# Final corrected code
number = 1
the_sum = 0

while number <= 100:
    the_sum = the_sum + number
    number += 1

print("The answer is", the_sum)
```

(b) aim: to find the median of a list of values

In [ ]:
```
def median(L):
    L_length = len(L)
    L_sorted = sorted(L)  # creates a new sorted list instead of doing L.sort

    if L_length % 2 = 1:  # odd number of elements     syntax error: a single
        return L_sorted[L_length/2]   # TypeError: index must be an integer a
    else:                 # even number of elements
        return (L_sorted[L_length/2] + L_sorted[L_length/2 + 1])/2   # logic

# Final corrected code
def median(L):
    L_length = len(L)
    L_sorted = sorted(L)

    if L_length % 2 == 1:   # use ==
        return L_sorted[L_length//2]   # use integer division (//) instead to
    else:
        return (L_sorted[L_length//2] + L_sorted[L_length//2 - 1])/2   # chan
```

(c) aim: to print and then remove all elements from a list

In [ ]:
```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# logic error: some values are skipped (2, 4, 6, 8)
for x in lst:
    print(x)
    lst.remove(x)

# Final corrected code
while lst:
    print(lst[0])
    lst.remove(lst[0])  # OR lst.pop(0)

# Always remember to use a while loop instead of a for loop when we want to c
```

Another debugging technique is **condition handling**, which allows us to take specific actions when **exceptions**, which are errors detected during the execution of a program, occur. The so-called handlers are programmed within the code to **catch exceptions**.

This can be done using `try - except (- finally )` statement.

In [ ]:
```
try:
    num = int(input("Enter a number: "))
except:
    print("Are you sure that is a number?")
finally:   # optional
    print("This is the end of the program.")
```

Our program can be made more specific in catching the different types of exceptions. The following are some common errors you may have committed along the way:

- **FileNotFoundError**
- **ZeroDivisionError**
- **TypeError**
- **ValueError**, e.g. typecasting a letter to an integer

In [ ]:
```
num1 = 1
num2 = 0

try:
    print(num1/num2)
except ZeroDivisionError as err1:
```

```
    print("An error occurs:", err1)
except TypeError as err2:
    print("An error occurs:", err2)
finally:
    print("This is the end of the program.")
```

Test Cases

In order to ensure the correctness of our program, it is important for us to design meaningful test cases to iron out any runtime and logic errors. In general, there are three types of test cases:

- normal (valid)
- boundary (extreme)
- abnormal (erroneous)

To understand each type of test cases, let us consider the following function to abbreviate a string of words separated by white spaces. For example, "Ministry of Education" should be abbreviated to "MOE" with the first letter of each word capitalised.

In [ ]:
```
# Taking care only normal test cases
def abbreviate(s):
    # Extract each word into a list
    lst = s.split()

    # Abbreviate and capitalise the first letter of each word
    result = ''

    for i in lst:
        result += i[0].upper()

    return result
```

(a) Normal test cases

First of all, we shall design test cases to make sure that the program to be tested works as intended under normal conditions, i.e. situations where the inputs are what we would expect to be provided with during normal use of the program. In general, under normal conditions, at least one test case should be designed for each type of expected input.

In the program discussed, only one type of input is expected: a string. Two examples are shown below.

In [ ]:
```
# Normal test case 1: The first letter of each word is in uppercase
# It should return "CS".
abbreviate("Computer Science")

# Normal test case 2: The first letter of each word can be in uppercase or lo
# It should return "MOE".
abbreviate("Ministry of Education")

# Normal test case 3: All letters are in lowercase
# It should return "PS".
abbreviate("political science")
```

(b) Boundary (extreme) test cases

Next, we look into boundary conditions, i.e. situations where the inputs are at the limits of what the program is designed for, or where special handling of data are required. The limits of a program are usually in terms of quantity and range (for `int` and `float`), as well as length (for iterables such as `str` and `list`). On the other hand, special handling is usually needed when the problem definition specifies that certain values are to be treated as exceptions, e.g. using -1 to represent missing data in a survey.

In the program discussed, perhaps there is little use in defining the maximum length for the inputs, so it may not be meaningful to design a test case that would be at the upper limit of what the program can handle. However, we do want the program to work as intended even with the minimum length of input possible, that is, an empty string. Also, what if we do not want to abbreviate a string that has only one word?

In [ ]:
```
# Boundary test case 1: Empty string
# It can be made to return "Nothing to abbreviate!"
abbreviate("")

# Boundary test case 2: A string with only one word
# It can be made to return "Nothing to abbreviate!"
abbreviate("Python")

# Modified code
# Can split according to a single white space and a dash
def split(s):
    lst = []
    temp = ""

    for char in s:
        if char == " " or char == "-": # char in [" ", "-"]
            if temp != "":
                lst.append(temp)
            temp = ""
        else:
            temp += char
    if temp != "":
        lst.append(temp)

    return lst

def abbreviate(s):
    if type(s) != str:
        return "Only strings can be abbreviated!"
    if len(s) <= 1:
        return "Nothing to abbreviate!"

    # Extract each word into a list
    lst = split(s)

    # Abbreviate and capitalise the first letter of each word
    result = ''

    for i in lst:
        if i.isnumeric():
            result += i
        else:
            result += i[0].upper()

    return result
```

We may also want to design test cases that differ slightly from their normal counterparts in

that they may be unexpected in the given context of the program, but are still valid inputs.

Suppose the programme discussed is intended only to abbreviate input strings that contain only the 26 letters in the English alphabets and a white space between each word. As such, abnormal test cases can be as follows.

In [ ]:
```
# Boundary test case 3: Extra white spaces
# It should return "MOE".
abbreviate(" Ministry of  Education ")

# Boundary test case 4: Inclusion of digits
# It should return "CS101".
abbreviate("Computer Science 101")

# Boundary test case 5: Anglo-Chinese Junior College
# It should return "ACJC".
abbreviate("Anglo-Chinese Junior College")

# How to handle the test cases above?
```

(c) Abnormal (erroneous) test cases

Finally, it is important to design test cases to ensure that the program runs smoothly under abonormal conditions, i.e. situations where the inputs would normally be rejected by the program.

In the program discussed, it is clear that we should only deal with input strings. Based on this criterion, we should return an appropriate error message when other data types are supplied.

In [ ]:
```
# Erroneous test case 1: Integer
# It can be made to return "Only strings can be abbreviated!"
abbreviate(123)

# Erroneous test case 2: Boolean
# It can be made to return "Only strings can be abbreviated!"
abbreviate(True)

# Modified code
def abbreviate(s):
    if type(s) != str:
        return "Only strings can be abbreviated!"
    elif len(s) <= 1:
        return "Nothing to abbreviate!"

    # Extract each word into a list
    lst = s.split()

    # Abbreviate and capitalise the first letter of each word
    result = ''

    for i in lst:
        result += i[0].upper()

    return result
```

Test cases for errors are necessary because programs that fail to reject invalid inputs may end up performing unintended or even harmful actions. For instance, a large number of security flaws in programs today are caused by the improper handling of error conditions. A typical example in some programming languages would be forgetting to check whether the

inputs supplied by a user can fit into the memory space allocated to store the data. A program that tries to store the invalid data anyway will end up overwriting subsequent areas of memory and allow potential attackers to insert their own instructions into the program.

2020 JC1 H2 Computing 9569

15. Data Validation and Verification

Data Validation

Data validation is a process to ensure that the data provided as inputs to programs conform with the requirements to avoid technical error. More often than not, a programmer does not have any control over the inputs supplied as they can come from many sources. Checks have to be done to make sure that the input data are acceptable.

Common data validation techniques are:

- range check
- format check
- length check
- presence check

(a) Range check

It is a check that limits an input to a particular range of values.

Here is an example of a code that takes in a test score as an integer between 0 and 100 inclusive, and prints out the grade.

In [ ]:
```
score = None
grade = ''

while score == None:
    score = int(input("Enter score: "))

    # Range check
    if score < 0 or score > 100:
        print("The score entered is out of range. It should be between 0 and
        score = None

if (score >= 70):
    grade = 'Distinction'
elif (score >= 60):
    grade = 'Merit'
elif (score >= 50):
    grade = 'Pass'
else:
    grade = 'Fail'

print("Grade: " + grade)
```

(b) Format check

It is a check that ensures that an input matches a required data type with a given format.

For instance, a particular form may require the date to be entered in a DD/MM/YYYY format. It needs to check that each part of the date has the correct number of digits. Checking that the month is between 01 and 12, and that the date is between 01 and 28, 29, 30, or 31 (depending on the month and year), would fall under the range check described above.

Consider the earlier code of converting a test score to a grade. If someone enters characters other than digits, the program will crash. As such, we need to handle such inputs.

In [ ]:
```
score = None
grade = ''

while (score == None):
    score = input("Enter score: ")

    # Format check
    if not score.isdigit():
        print("You have entered an invalid score. It must be an integer betwe
        score = None
        continue

    score = int(score)

    # Range check
    if (int(score) < 0 or int(score) > 100):
        print("The score entered is out of range. It should be between 0 and
        score = None

if (score >= 70):
    grade = 'Distinction'
elif (score >= 60):
    grade = 'Merit'
elif (score >= 50):
    grade = 'Pass'
else:
    grade = 'Fail'

print("Grade: " + grade)
```

(c) Length check

It is a check that limits an input to a certain (range of) length.

Below is a simple code to check if a newly entered password is at least 8 characters long.

In [ ]:
```
password = None

while password == None:
    password = input("Create a password: ")

    # Length check
    if len(password) < 8:
        print("Your password needs to be at least 8 characters long.")
        password = None

print("Password accepted.")
```

(d) Presence check

It is a check that ensures that a required input is supplied.

Going back to the simple new password checker mentioned above, we shall display an error message if no password is supplied.

In [ ]:
```
password = None

while password == None:
    password = input("Create a password: ")
```

```
    # Presence check
    if (password == ''):
        print("You have not entered anything.")
        password = None

    # Length check
    elif len(password) < 8:
        print("Your password needs to be at least 8 characters long.")
        password = None

print("Password accepted.")
```

In summary, data validation ensures that the data are of the correct type and format. However, it does not ensure that the data are accurate.

Data Verification

Data verification is a way to confirm that the data entered was what was intended to be entered.

Common data verification techniques are:

- check digit
- double entry
- proofreading data

(a) Check digit

It is a piece of information used to detect typos in data, especially for sensitive ones such as registration numbers, car numbers, etc.

For example, every published book has an ISBN-13 number (International Standard Book Number), which is a 13-digit number usually printed above or below the barcode. This allows publishers, retailers and libraries to be able to know exactly which edition of a book they are referring to without ambiguity.

Suppose the 13 digits are labelled x_1 to x_{13}. The **weighted sum**

$$s = (x_1 + 3x_2 + x_3 + 3x_4 + x_5 + 3x_6 + x_7 + 3x_8 + x_9 + 3x_{10} + x_{11} + 3x_{12} + x_{13})$$

is calculated.

A valid ISBN-13 number must satisfy

```
s % 10 == 0
```

For instance, the ISBN-13 number of *Cambridge International AS and A Level Computing Coursebook* is 978-0-521-18662-9.

$$s = (9 + 3 \times 7 + 8 + 3 \times 0 + 5 + 3 \times 2 + 1 + 3 \times 1 + 8 + 3 \times 6 + 6 + 3 \times 2 + 9) = 100$$

and indeed

```
100 % 10 == 0
```

is true.

If there were an error in any one digit, the sum would not be divisible by 10. This provides a small measure of checking against transmission errors as it would need at least two errors to make the (wrong) sum divisible by 10.

This procedure is known as **modulo-10 weighted check digit calculation**.

Before 2007, the ISBN number was only 10 digits long (there was no 978 or 979 in front) and the check digit used a **modulo-11 weighted check digit calculation**. More details are given in the tutorial.

More details on the ISBN system *(not in syllabus)*

Referring to the same example of 978-0-521-18662-9, the 13 digits are broken down in the following way:

- The first three digits are always 978 or 979 for books. Other sets of three digits are used for other consumer products.
- The next digit, 0, is called the group. It refers roughly to the language area where the book is published. 0 and 1 are used for English-speaking countries.
- The next three digits, 521, identifies the publisher, Cambridge University Press.
- The next five digits, 18662, refers to the specific book and edition.
- The last digit, 9, is the check digit, which is chosen to make the formula above true.

(b) Double entry

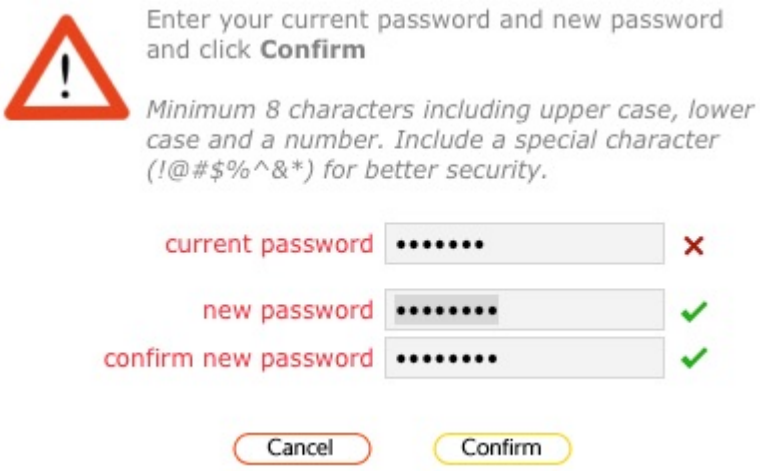
It is a process that asks a user to enter the required data twice. This is commonly used, for instance, with passwords or e-mail addresses, where a user might easily type one character wrongly. It is even more pertinent with passwords because the characters are usually not displayed on the screen while the user is typing.

(c) Proofreading data

This can take place in several ways. For instance, the user may be asked to do a visual check of what has been entered before confirming submission. This is used in some online forms, where users are asked to check what they have entered one more time before the final submission.

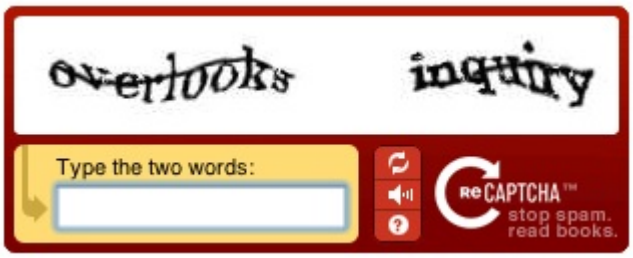
Another example of proofreading data is is one where the entered data is compared with data that already exists in a database.

As an example, take a look at the image below that shows a page that is commonly seen when we want to change an account password.



Double entry is employed to change the password, where the user is required to enter the new password twice. Proofreading data, on the other hand, ensures that a user enters his old password as stored in the database correctly before the new password is accepted. Should the old password be entered wrongly, an error message such as "old password is incorrect" is displayed.

In today's era of illegitimate use of automation, we see increasingly more and more websites employing the use of **Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA)**, a program implemented to thwart spam and automated extraction of data from a particular system. CAPTCHAs can be considered as a data verification process to ensure that the required fields are keyed in by human beings.



2020 JC1 H2 Computing 9569

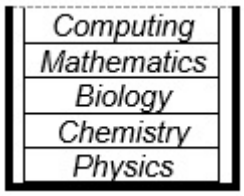
16. Stack and Queue

Building on our existing knowledge of OOP, we shall consider two simple, but powerful concepts that are familiar to us in our daily lives.

Stack and **queue** are two linear data structures whose items are ordered according to how they are added or removed.

Stack

Consider a stack of five books in a container as shown below.



To come up with the stack, one has to put Physics first into the container, which forms the base of the stack. Subsequently, he has to put Chemistry on top of Physics, followed by Biology, Mathematics and finally Computing, which is now at the top of the stack and is readily accessible. In order to retrieve Biology, for example, he has to take out Computing and Mathematics first in that order.

From the simple illustration above, we can see that most recently added item to the stack is the one that is in the position to be removed first. The stack data structure has this ordering principle known as **last-in-first-out (LIFO)**.

There are other examples of stacks in everyday situations. A stack of trays is a common sight at any food stalls in a hawker centre. One takes the tray from the top of the stack to carry the food items purchased, uncovering the next tray for another customer in line. Can you think of more examples?

The table below shows the common methods of the stack data structure.

Creating the Stack Class

Refer to the sample program given below.

Method	Return	Use
is_empty()	boolean	checks whether the stack is empty
size()	integer	counts the number of items in the stack
push(item)	none	adds an item to the top of the stack
pop()	item	removes and returns the item at the top of the stack
peek()	item	shows the item at the top of the stack

We shall now try to create a *Stack* class with the following methods:

- is_empty()
- size()
- push(item)
- pop()
- peek()

When initialised, the *Stack* object should be empty.

In [58]:
```
# Type your code here
class Stack:
    def __init__(self): self.stack = []
    def is_empty(self): return not self.stack
    def size(self): return len(self.stack)
    def push(self, item): self.stack.append(item)
    def pop(self): return self.stack.pop() if self.stack else
None
    def peek(self): return self.stack[-1] if self.stack else
None
```

Run the following codes to see if your implementation works.

In [59]:
```
s = Stack()
```

In [60]:
```
# This should display True
print(s.is_empty())
```

```
True
```

In [61]:
```
# There is no item to peek
print(s.peek())
```

```
Stack is empty!
```

In [62]:
```
s.push(2)
```

In [63]:
```
s.push(5)
```

In [64]:
```
#This should display 5
print(s.peek())
```

```
5
```

In [65]:
```
s.push('b')
```

In [66]:
```
# This should display 3
print(s.size())
```

```
3
```

In [67]:
```
# This should display 'b'
print(s.pop())
```

```
b
```

In [68]:
```
# This should display 5
print(s.pop())
```

```
5
```

In [69]:
```
# This should return False
print(s.is_empty())
```

```
False
```

Use of Stack

A stack can be used in backtracking problems. Imagine a program that has to find its way through a maze. Chances are the program will come to an intersection, choose one direction and continue down that path. If it hits a dead end, how does it know what to do next? Each decision point can be pushed to a stack. If the outcome is not the expected one, the last decision point can be popped off and another path can be traversed. You will do this as one of the missions in Coursemology.

A more complex task is recursive descent parsing in Natural Language Processing. To put it simply, the program traverses a tree data structure, following a given grammatical path. During the traversal, the path is saved in a stack that records each movement that is made.

Queue

One of the simplest example of a queue is the typical line that we all participate in from time to time, such as at a cinema counter to get a movie ticket or at a check-out lane in a supermarket. Someone at the head of the queue is going to be served and removed from the queue first, while a new person who wants to join the queue has to go to the tail of the queue. Contrary to the stack data structure, this ordering principle is known as **first-in-first-out (FIFO)**.

In another scenario, imagine a printing queue within a computer laboratory with 30 computers connected to a single printer via a local network. When students want to print something out, their print tasks “get in line” with the rest. The first student to send his printing job shall receive the printout first, while the last person to do so has to wait for all the other printing tasks to be completed. The actual implementation, however, uses spooling, which is more complicated than a proper queue.

The table below shows the common methods of the queue data structure.

Code	Stack Content	Return

s = Stack()	[ ]	[ ]
s.is_empty()	[ ]	True
s.peek()	[ ]	
s.push(2)	[2,]	
s.push(5)	[2, 5]	
s.peek()	[2, 5]	5
s.push('b')	[2, 5, 'b']	
s.size()	[2, 5, 'b']	3
s.pop()	[2, 5]	'b'
s.pop()	[2,]	5
s.is_empty()	[2,]	False

Creating the Queue Class

Refer to the sample program given below.

Method	Return	Use
is_empty()	boolean	checks whether the queue is empty
size()	integer	counts the number of items in the queue
enqueue(item)	none	adds an item to the tail of the queue
dequeue()	item	removes and returns the item at the head of the queue
show_head()	item	shows the item at the head of the queue
show_tail()	item	shows the item at the tail of the queue

We shall now try to create a *Queue* class with the following methods:

- is_empty()
- size()
- enqueue(item)
- dequeue()
- show_head()
- show_tail()

When initialised, the *Queue* object should be empty.

In [1]:
```
# Type your code here
class Queue:
    def __init__(self): self.queue = []
    def is_empty(self): return not self.queue
    def size(self): return len(self.queue)
    def enqueue(self, item): self.queue.append(item)
    def dequeue(self): return self.queue.pop(0) if self.queue
else None
    def show_head(self): return self.queue[0] if self.queue
else None
    def show_tail(self): return self.queue[-1] if self.queue
else None
```

Run the following codes to see if your implementation works.

In [103…
```
q = Queue()
```

In [104…
```
# This should return True
print(q.is_empty())
```

```
True
```

In [105…
```
# There is no item at the head
print(q.show_head())
```

```
Stack is empty!
```

In [106…
```
q.enqueue(3)
```

In [107…
```
q.enqueue(8)
```

In [108…
```
# This should display 3
print(q.show_head())
```

```
3
```

In [109…
```
# This should display 8
print(q.show_tail())
```

```
8
```

In [110…
```
# This should display 2
print(q.size())
```

```
2
```

In [111…
```
q.enqueue('k')
```

In [112…
```
# This should display 'k'
print(q.show_tail())
```

```
k
```

In [113…
```
# This should display 3
print(q.dequeue())
```

```
3
```

Use of Queue

A queue is used in breadth-first search, a type of search algorithm used in a tree data structure.

There are other variations, such as circular queue and double-ended queue. For one, the former is used in memory management and process scheduling.

As a heads up, you will use the queue data structure to perform radix sort, a non-comparative sorting algorithm, as one of the missions in Coursemology.

Function	Queue Content	Return
q = Queue()	[ ]	[ ]
q.is_empty()	[ ]	True

q.show_head()	[ ]	
q.enqueue(3)	[3]	
q.enqueue(8)	[3, 8]	
q.show_head()	[3, 8]	3
q.show_tail()	[3, 8]	8
q.size()	[3, 8]	2
q.enqueue('k')	[3, 8, 'k']	
q.show_tail()	[3, 8, 'k']	'k'
q.dequeue()	[8, 'k']	3

2020 JC1 H2 Computing 9569

17. Arrays, Static Memory and Pointers

Python is a high-level language, meaning that many lower-level processes are automated into single functions or methods. This also means that some of the data types used by Python have features that are more advanced than those of other languages. While these features make Python easier to learn and use, they also mean that Python runs more slowly than other languages. (You are not going to notice this with the data encountered in school because it is a difference of fractions of a second. However, if you are dealing with actual big data containing thousands, or even millions, of entries, this will be significant.) One such data type used by Python is the list.

Arrays

In some other languages, a data type known as the **array** is used. This is a list of items, usually (though not always) of the same data type. The main difference between an array and Python's list is that when an array is first declared, it has a **fixed length**. This means that while the programmer can make changes to individual items in the array, the programmer cannot delete or append items to change the length of the array.

While Python does not have the array data type, we can simulate it using OOP. The code below does the following:

1. Create an array, of a given length, of strings.
2. Changing the entry at a certain index.
3. Returning the entry at a certain index.

In [1]:
```
class Array:

    def __init__(self, n):
    # The array is initialised as a list, of length n, of empty strings.
        self.Array = []
        for i in range(n):
            self.Array.append("")

    def add_entry(self, i, newstring):
    # Change the entry at index i to newstring
        self.Array[i] = newstring

    def get_entry(self, i):
    # Returns the entry at index i
        return self.Array[i]

    # If we want to make a delete method, the best we can do is
    # make a method to replace the string there with an empty string.

    def delete_entry(self, i):
    # Deletes the entry at index i
        self.add_entry(i, "")

MyArray = Array(10)
MyString = "singapore"
```

```
for i in range(len(MyString)):
    MyArray.add_entry(i, MyString[:i])

for i in range(4,7):
    print(MyArray.get_entry(i))

MyArray.delete_entry(5)

for i in range(4,7):
    print(MyArray.get_entry(i))
```

```
sing
singa
singap
sing

singap
```

While Python (and Java) starts the index numbering from 0, there is no actual fixed convention about whether the first number should be 0 or 1. In your A-level examination, both conventions may be used. The code below shows the same array as above, but now with indices 1 to n (inclusive).

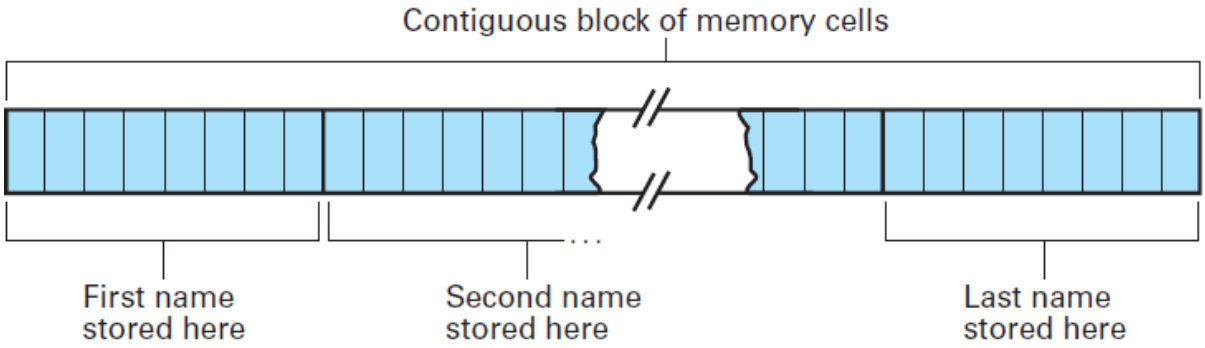
In [ ]:
```
class Array:
    def __init__(self, n):
        self.Array = []
        for i in range(n):
            self.Array.append("")

    def add_entry(self, i, newstring):
        self.Array[i-1] = newstring

    def get_entry(self, i):
        return self.Array[i-1]
```

In the event that the data is best represented in a table, each entry in an array can be an array (or list) itself, representing the entries in each row of the table. The main array then becomes a list of the rows. This is known as a **two-dimensional array**.

In [3]:
```
class TDArray:
    def __init__(self, row, col):
        self.TDArray = []
        for _ in range(row):
            temp = [""] * col
            self.TDArray.append(temp)

    def add_entry(self, row, col, new_string):
        self.TDArray[row][col] = new_string

    def get_entry(self, row, col):
        return self.TDArray[row][col]

    def delete_entry(self, row, col):
        self.add_entry(row, col, "")
```

Static Memory

In a computer's memory, a list of entries may be stored as a single continuous block of memory cells with consecutive addresses. We need to determine how much space to allocate to each entry, as well as the number of entries, to determine how much memory to allocate to the entire list. This way of organising the memory is called a **contiguous list**.



The array we have described previously provides us a way to *simulate* data in the computer's memory, with the entries of the array corresponding to cells in the memory block. We can only access or change cells using their index. However, this is not always the most convenient way to do things. For example, if we wished to print out all the non-empty entries in the block, we would have to iterate through the entire block, possibly skipping over the empty entries.

One solution is to always move entries so that the space occupied by the data within the block remains contiguous. If the list is a static list (i.e. it is not going to change), this storage structure is a convenient one. However, in the case of a dynamic list that changes frequently, after we delete an entry, we need to move all the following entries up by one index. If we add an entry in the middle, we need to first move all the following entries down by one index.

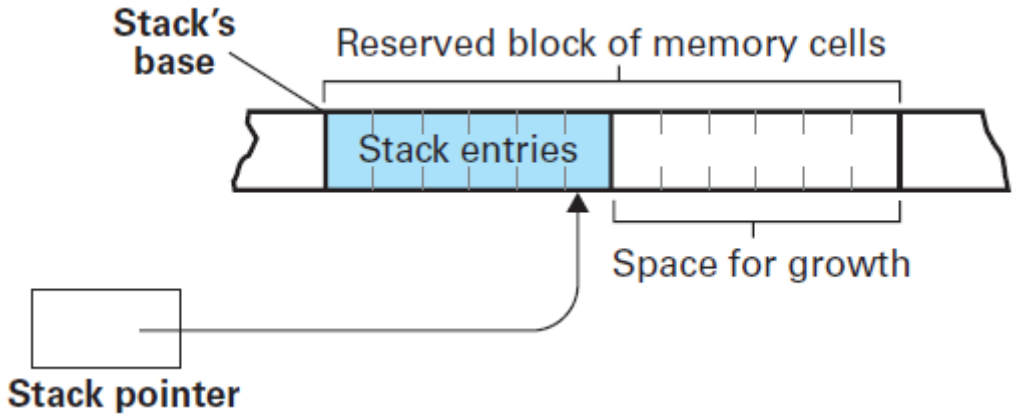
To avoid this, we can restrict ourselves further - we only allow ourselves to add entries at one end of the array, and we only allow ourselves to delete entries at the other end of the array.

- If we can add and delete entries at the same end, our array becomes a **stack**.
- If we add entries at one end and delete them at the other end, our array becomes a **queue**.

One additional concept is required. Our data may be contiguous, but we would like to avoid having to iterate through the entire array to find where it starts and where it ends. Therefore, we introduce a variable called a **pointer** to tell us the location of one of the ends of the data. As we add and delete entries, we would have to update the value of the pointer so that it always points to the first or last entry.

Stack (Revisited)

A memory block large enough to accommodate the expected maximum size of the stack is reserved. (How large this is depends on what the stack will be used for. This is not always easy to determine.) One end of the memory block is designated as the base, and the entries, or nodes, get pushed onto the stack next to each other. The stack thus grows towards the other end of the reserved memory block.



Since the bottom of the stack is always at the beginning of the memory block, we do not need to have a pointer there as we know where it is. However, as entries get pushed and popped, the top of the stack moves back and forth within the memory block. To keep track of this location, we use a pointer called the **stack pointer**.

The stack is thus initialised as an array of empty cells.

To push:

1. Move the stack pointer to point to the empty cell next to the top of the stack.
2. Place the new entry at this location.

To pop:

1. Read (and delete) the data at the pointer.
2. Move the stack pointer to the next entry of the stack.

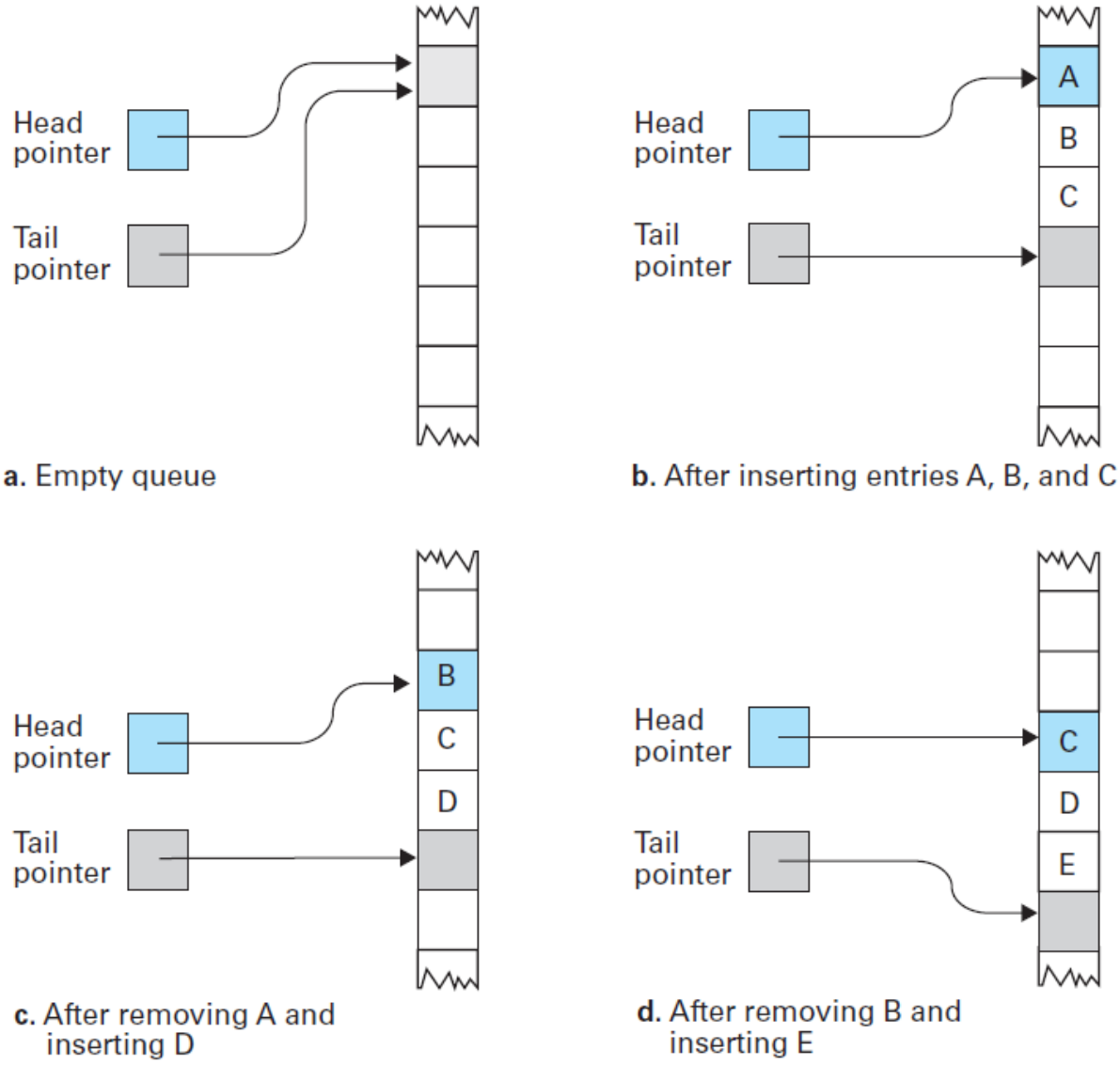
In [ ]:
```
class Stack(Array):

    def __init__(self, n):

        # We define a Stack as a subclass of Array.
        # In this example, the convention for the indices is that
        # they go from 0 to n-1 (inclusive).

        self.Array = []
        for i in range(n):
            self.Array.append("")

        self.StackPointer = 0
        # In an empty Stack, the StackPointer points to a non-existent entry.
        # Note that in the code, we do not use -1 to refer to the
        # last entry of the array.

    def size(self):

        # This returns the number of elements currently in the Stack.

        return self.StackPointer

    def push(self, newstring):

        # The stack is full when there are n entries in it.
        # The StackPointer then points to the hypothetical cell at index = n

        if self.StackPointer == len(self.Array)-1:
            print("Error: Stack is full!")
```

```
        else:
            self.StackPointer += 1
            self.add_entry(self.StackPointer, newstring)

    def pop(self):

        # The stack is empty when there are no entries in it.
        # The StackPointer then points to hypothetical cell at index = -1.

        if self.StackPointer == -1:
            print("Error: Stack is empty!")
        else:
            data = self.Array[self.StackPointer]
            self.Array[self.StackPointer] = ""
            self.StackPointer -= 1
            return data

    def peek(self):
        if self.StackPointer == -1:
            print("Error: Stack is empty!")
        else:
            return self.Array[self.StackPointer]
```

Queue (Revisited)

A queue is similar in implementation to a stack. Again, a block of memory sufficiently large to accommodate the expected maximum size of the queue is reserved. For a queue, we need pointers at both ends, which requires two additional memory cells, one for the **head pointer** and one for the **tail pointer**.

In an empty queue, both the head and tail pointer would point to the same location. Otherwise, the head pointer points to first entry in the queue, and the tail pointer points to the empty cell after the last entry of the queue.



a. Empty queue

b. After inserting entries A, B, and C

c. After removing A and inserting D

d. After removing B and inserting E

To enqueue:

1. Write the data in the empty cell at the tail pointer's location.
2. Move the tail pointer to the next cell.

To dequeue:

1. Read (and delete) the data at the head pointer.
2. Move the head pointer to the next entry in the queue.

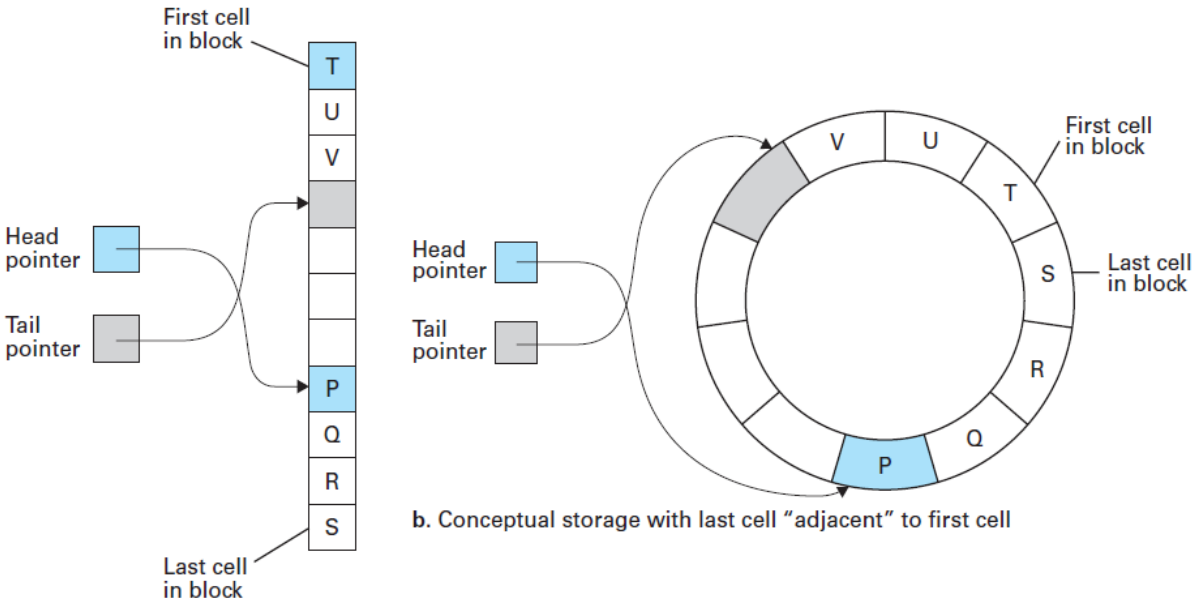
You will implement a linear queue in your tutorial.

Circular Queue

One problem with the queue described above is that, as entries are enqueued and dequeued, the queue itself will slide through the memory block. It might eventually reach the end of the reserved block, but have many empty cells in front of the head end of the queue. To keep this from happening, we implement what is called a **circular queue**.

When the tail of the queue reaches the end of the block, new entries are enqueued at the beginning of the block, which should be empty at this time. When elements have been dequeued until the head of the queue reaches the end of the block, the head pointer is also

redirected to be beginning of the block. In an abstract sense, the ends of the block are linked together in a circle (though this does not actually happen physically).



The code below shows one way to implement a circular queue.

In [ ]:
```
class CircularQueue(Array):
    def __init__(self, n):

        # We define a CircularQueue as a subclass of Array.
        # In this example, the convention for the indices is that
        # they go from 0 to n-1 (inclusive).

        self.Array = []
        for i in range(n):
            self.Array.append("")

        self.Head = 0
        self.Tail = 0
        # The head and tail pointers both start off pointing to
        # the smallest index in the array.

        self.MaxSize = n-1
        # Since the tail pointer must always point to an empty cell,
        # we can only put a maximum of n-1 entries in the queue.

    def size(self):

        # This returns the number of elements currently in the queue.
        # This is useful later.

        if self.Tail >= self.Head:
            return (self.Tail - self.Head)
        else:
            # If the tail has gone around the circle but the head has not,
            # what is the size of the queue?
            return (self.MaxSize + 1 - (self.Head-self.Tail))

    def enqueue(self, newstring):

        # The CircularQueue is full when there are n-1 entries in it.

        if self.size() == self.MaxSize:
            print("Error: Queue is full!")
```

```
        else:
            self.add_entry(self.Tail, newstring)
            self.Tail = (self.Tail + 1) % (self.MaxSize + 1)

    def dequeue(self):

        # The circular queue is empty when there are no entries in it.
        # The head and tail pointers are thus pointing to the same cell.

        if self.size() == 0:
            print("Error: Queue is empty!")
        else:
            data = self.Array[self.Head]
            self.Array[self.Head] = ""
            self.Head = (self.Head + 1) % (self.MaxSize + 1)
            return data

    def readHead(self):
        if self.size() == 0:
            print("Error: Queue is empty!")
        else:
            return self.Array[self.Head]

    def readTail(self):
        if self.size() == 0:
            print("Error: Queue is empty!")
        else:
            return self.Array[self.Tail-1]
            # Here, we are using Python's convention that when the index is -
            # it refers to the last entry in the array.
```

2020 JC1 H2 Computing 9569

18. Linked List and Dynamic Memory

The order of data entries in the array corresponds directly to the order in which the entries are stored in the memory block of the computer. As we have seen, we would like to keep the data in the array contiguous so that it is easy to iterate through the data to locate the entry that we want. For an array where data gets inserted and deleted frequently, this is troublesome as it would mean having to shuffle data around repeatedly.

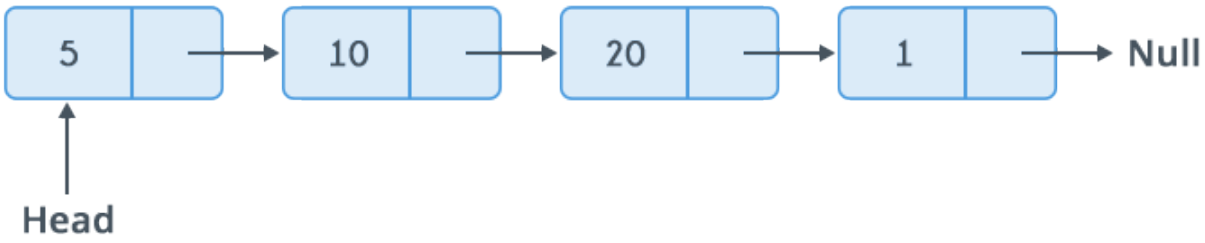
In the previous chapter, we saw that one way to get around this problem is to prevent it from arising, i.e., to add or delete data only at one end of the array. This led us to two data structures, the stack and the queue. We also introduced the concept of the pointer.

Using pointers, we can also solve the problem of having a frequently-changing array in a different way. This is to make use of a new data structure, a **linked list**.

A linked list is a linear data structure where the order of the data entries may not correspond their physical placement in the memory block of the computer. In a linked list, each entry may be stored in a different area of the memory block. However, we still need to ensure that we can still go from one entry to the next in a list. To do this, each entry should be represented as a **node**, which consists of two parts: the **data**, and a **pointer** (typically called **next**) to indicate the location of the next entry in the list. (You may think of a node as a list/tuple of length two.)

We also need to know where the list begins and ends. For the beginning of the list, we have a pointer to indicate the location of the first entry, known as the **start pointer** or **head pointer**. At the end of the list, the last entry has a **null pointer** (in the case of Python, the pointer shall point to `None`). This indicates that there are no further entries in the list.

The diagram below shows how the list `[5, 10, 20, 1]` looks like as a linked list.



Creating a Linked List

Before we can create the `LinkedList` class, we need to define the `Node` class, the objects of which shall be the buildings blocks for our linked list.

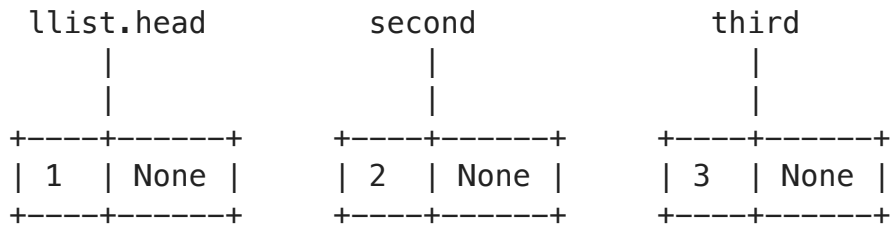
In [ ]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
```

With the above, we can now proceed to make a linked list `llist` to represent the list `[1, 2, 3]`.

In [ ]:
```
llist = LinkedList()

llist.head  = Node(1)
second = Node(2)
third  = Node(3)
```

Three nodes have been created, namely `llist.head`, `second` and `third`. At the moment, they are not linked to one another as all their pointers point to `None`.

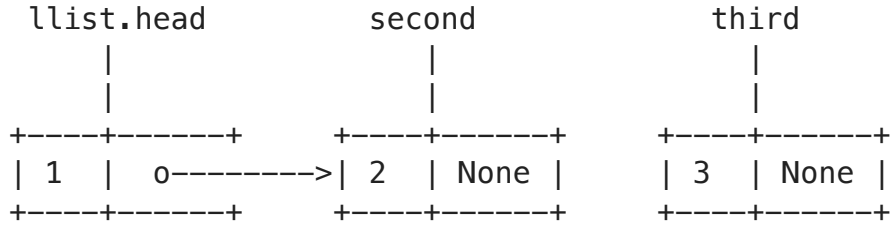
In [ ]:
```
print(llist.head.next)
print(second.next)
print(third.next)
```



Let us link the first node with the second.

In [ ]:
```
llist.head.next = second

print(llist.head.next)
print(llist.head.next.data)
```

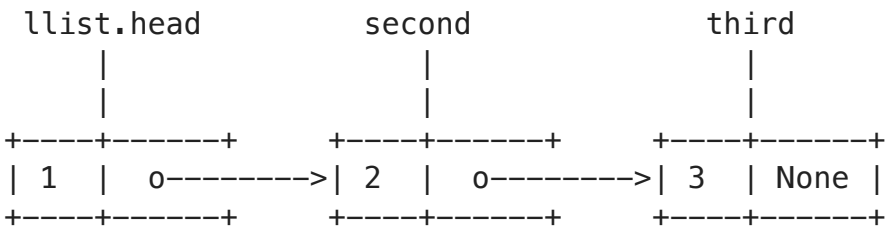
The first node has been linked to the second.



How should we link the second node with the third?

In [ ]:
```
print(second.next)
print(second.next.data)
```

We have now successfully created the linked list with three entries.



Printing Entries

The method `PrintList` contains a variable called `temp` that helps us to iterate through the entries and prints them out sequentially.

In [6]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def PrintList(self):
        temp = self.head
        while temp != None:
            print(temp.data)
            temp = temp.next
```

Try running the code below to check whether your implementation of `PrintList` is correct.

In [ ]:
```
llist = LinkedList()

llist.head  = Node(1)
second = Node(2)
third  = Node(3)

llist.head.next = second
second.next = third

llist.PrintList()
```

Adding a Node

Notice that there are three places where we can add nodes:

- at the beginning (`Push` method)
- somewhere in the middle (`InsertAfter` method)
- at the end (`Append` method)

You may wish to use the space below to draw a diagram to illustrate what the code is actually doing.

In [ ]:

```
class LinkedList:
    # Copy and paste the methods: __init__, printList

    def Push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def InsertAfter(self, prev_data, new_data):
        #check if prev_node is in the linked list
        temp = self.head
        while temp != None:
            if temp.data == prev_data:
                new_node = Node(new_data)
                new_node.next = temp.next
                temp.next = new_node
                break
            else:
                temp = temp.next
        if temp == None:
            # iterated thru whole list, cannot find prev_data
            print("Error: Cannot find prev_data")

    def Append(self, new_data):
        new_node = Node(new_data)

        # if the linked list is empty, new_node shall become the first enti
        if not self.head:
            self.head = new_node
            return

        # at the end of the loop, last points to the last node before appe
        last = self.head

        while last.next:
            last = last.next

        last.next = new_node
```

Work through the code below and try to guess the sequence of the entries in the linked list before running it.

In [ ]:

```
llist = LinkedList()

llist.Append(6)
llist.Push(7)
llist.Push(1)
llist.Append(4)
llist.InsertAfter(7, 8)
llist.InsertAfter(8, 9)

print('Result:')
llist.PrintList()
```

At this point you may be wondering if it is possible to `InsertAfter` a node with a given index (in the linked list). You would need to modify the code of `InsertAfter` to do this.

You may also be wondering if it is possible to `InsertAfter` a node with a given data. To do this, we would first need to search for the node with that particular data within the list. This is addressed both of the sections below, "Deleting a Node" and "Searching for an Entry".

Deleting a Node

We may identify the node to be deleted by its index or by its data. The code below shows how to identify a node by its data before deleting it.

What are some considerations we must bear in mind before deleting a node?

In [ ]:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    # Copy and paste the methods: __init__, PrintList, Push, InsertAfter, A

    def DeleteNode(self, to_delete):
        temp = self.head

        # if the list if empty, there is nothing to be done
        if not self.head:
            return

        # if the first node is the one to be deleted
        if self.head.data == to_delete:
            self.head = self.head.next
            temp = None                # delete the data in temp
            return

        # when the node to be deleted is somewhere else
        # need to know the node BEFORE the one to be deleted when the loop
        while temp:
            if temp.data == to_delete:
                break
            prev = temp
            temp = temp.next

        prev.next = temp.next
        temp = None                    # delete the data in temp
```

Try running the code below to check whether your implementation of `deleteNode` is correct.

In [ ]:

```
llist = LinkedList()
llist.Push(7)
llist.Push(1)
llist.Push(3)
llist.Push(2)

llist.PrintList()

llist.DeleteNode(1)
llist.PrintList()
```

Searching For an Entry

Searching for an entry - that is, searching for a node which has a given data stored inside it - can either be done iteratively or recursively.

In [ ]:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    # Copy and paste the methods: __init__, printList, push, insertAfter, a

    def Search_iter(self, x):       # iterative search
        temp = self.head
        while temp.data != x:
            temp = temp.next
            if temp == None:
                return "Not Found!"
        return "Found!"

    def Search_recur(self, x):  # recursive search
        def Search_recur_helper(cur, x):
            if not cur:
                return False
            elif cur.data == x:
                return True
            else:
                return Search_recur_helper(cur.next, x)

        return Search_recur_helper(self.head, x)
```

Either search method should return `True` when the integer `21` is supplied as an argument.

In [ ]:

```
llist = LinkedList()

llist.Push(10)
llist.Push(30)
llist.Push(11)
llist.Push(21)
llist.Push(14)

print(llist.Search_iter(21))
print(llist.Search_recur(21))
```

Setting Up a Linked List Inside an Array

Similar to what we have done for stacks and queues, we would like to simulate a computer's memory block using an array and use that to store a linked list. To do that, we need to first create something called a **free list**. This is a linked list containing all the unused memory cells. At the beginning, since all the cells are unused, the free list would be the same as the array.

In [ ]:

```
class Array:

    # The array has indices 0 to n-1 (inclusive).
    # Each entry in the array is a node, which will be a list of two elemer

    def __init__(self, n):
        self.Array = []
        for i in range(n-1):
            node = ["", i+1]

            # node[0] the data, which we initialise using an empty string.
            # node[1] is the pointer.

            self.Array.append(node)

        self.Array.append(["",None])

        self.FreeListPointer = 0 # Head pointer of free list.
```

Our array now looks like this (where we use `n=5`):

```
FreeListPointer = 0

Array:
    +------+---------+
    | Data | Pointer |
    +------+---------+
[0] |  ""  |    1    |
[1] |  ""  |    2    |
[2] |  ""  |    3    |
[3] |  ""  |    4    |
[4] |  ""  |  None   |
    +------+---------+
```

If we were to visualise this linked list using the diagrams we had above, it would look like this:

```
FreeListPointer
     |
     |
+----+-----+   +----+-----+   +----+-----+   +----+-----+
+----+------+
| "" |  o------>| "" |  o----->| "" |  o----->| "" |  o----->|
"" | None |
+----+-----+   +----+-----+   +----+-----+   +----+-----+
+----+------+
(Cell [0])     (Cell [1])     (Cell [2])     (Cell [3])
(Cell [4])
```

Suppose we decide to create a linked list using this array. We would begin by initialising it as an empty linked list (i.e. the head pointer points to `None`).

To add a node to the linked list:

1. Determine where the node is to be added (see "Adding a Node" abvove).
2. The data for the new node goes into the first node of the free list.
3. Update the pointers (not necessarily in the order indicated):
    A. Free list pointer points to the second node of the free list.
    B. The new node points to the node after the insertion point.
    C. The node before the insertion point points to the new node.

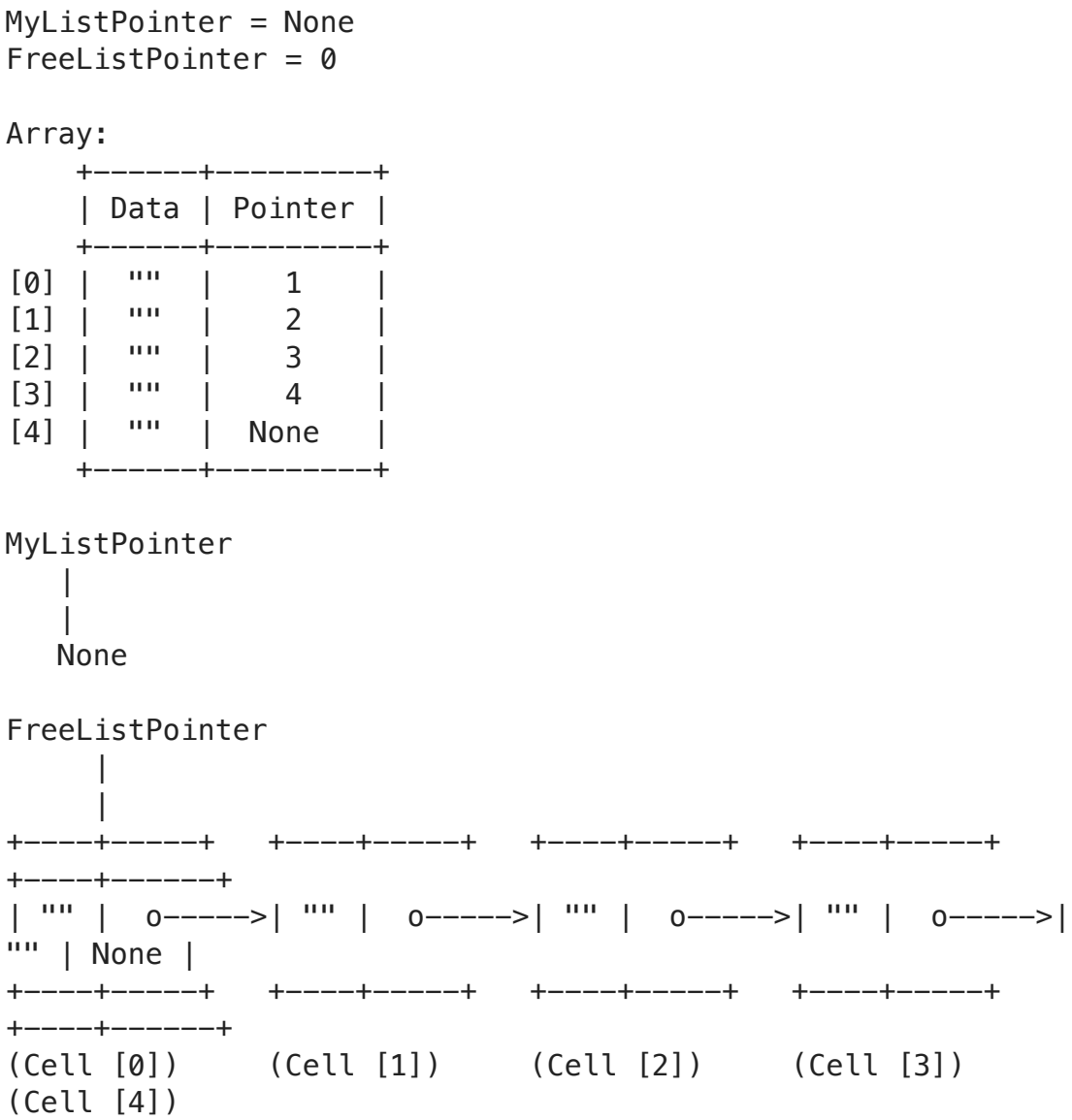
To delete a node from the linked list:

1. The data of the node is deleted and the node is added to the beginning of the free list.
2. The pointers in the linked list are updated.

Example

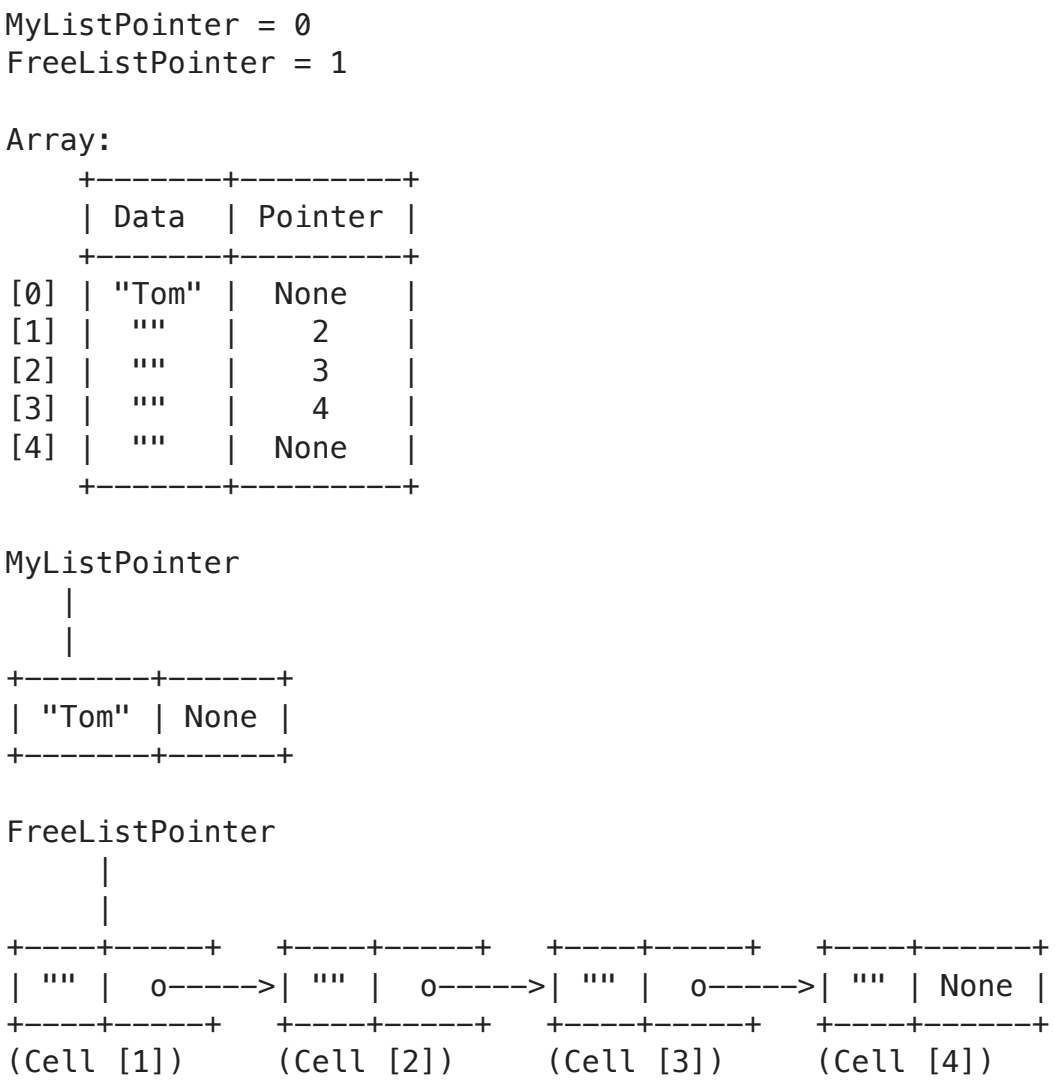
Step 1

We create a list called `MyList`. Initially it is empty.



Step 2

We add `"Tom"` into `MyList`.



Step 3

We add `"Sue"` into `MyList` before `"Tom"`.

```
MyListPointer = 1
FreeListPointer = 2

Array:
    +-------+---------+
    | Data  | Pointer |
    +-------+---------+
[0] | "Tom" |  None   |
[1] | "Sue" |    0    |
[2] |  ""   |    3    |
[3] |  ""   |    4    |
[4] |  ""   |  None   |
    +-------+---------+

MyListPointer
   |
   |
+-------+-----+   +-------+------+
| "Sue" |  o----->| "Tom" | None |
+-------+-----+   +-------+------+

FreeListPointer
     |
     |
+----+-----+   +----+-----+   +----+------+
| "" |  o----->| "" |  o----->| "" | None |
+----+-----+   +----+-----+   +----+------+
(Cell [2])     (Cell [3])     (Cell [4])
```

Step 4

We add `"Ted"` into `MyList` after `"Sue"`.

```
MyListPointer = 1
FreeListPointer = 3

Array:
    +-------+---------+
    | Data  | Pointer |
    +-------+---------+
[0] | "Tom" |  None   |
[1] | "Sue" |    2    |
[2] | "Ted" |    0    |
[3] |  ""   |    4    |
[4] |  ""   |  None   |
    +-------+---------+

MyListPointer
   |
   |
+-------+-----+   +-------+-----+   +-------+------+
| "Sue" |  o----->| "Ted" |  o----->| "Tom" | None |
+-------+-----+   +-------+-----+   +-------+------+

FreeListPointer
     |
     |
+----+-----+   +----+------+
| "" |  o----->| "" | None |
+----+-----+   +----+------+
(Cell [3])     (Cell [4])
```

Step 5

We delete `"Sue"` from `MyList`.

```
MyListPointer = 2
FreeListPointer = 1

Array:
    +-------+---------+
    | Data  | Pointer |
    +-------+---------+
[0] | "Tom" |  None   |
[1] |  ""   |    3    |
[2] | "Ted" |    0    |
[3] |  ""   |    4    |
[4] |  ""   |  None   |
    +-------+---------+

MyListPointer
   |
   |
+-------+-----+   +-------+------+
| "Ted" |  o----->| "Tom" | None |
+-------+-----+   +-------+------+

FreeListPointer
     |
     |
+----+-----+   +----+-----+   +----+------+
| "" |  o----->| "" |  o----->| "" | None |
+----+-----+   +----+-----+   +----+------+
(Cell [1])     (Cell [3])     (Cell [4])
```

Notice that neither `MyList` nor the free list forms a contiguous block of cells in the array. Also notice that the order of data entries in `MyList` is different in the array, and in the linked list itself.

Instead of moving data around within the array, we leave it in the fixed location and only update pointer values.

Note:

1. Some programmers use `-1` (instead of `None`) to indicate the null pointer. In this case, we need to avoid writing code in Python where `-1` is used as the index of the last element of a list to avoid confusion.

2. In the example shown, our array indices are from `0` to `n-1`. Cambridge may also use array indices `1` to `n`. In this case, it is common to use `0` for the null pointer.

3. Notice that it is possible for two (or more) linked lists to share the same array. One example is shown below.

```
BoyListPointer = 2
GirlListPointer = 3
FreeListPointer = 1

Array:
    +-------+---------+
    | Data  | Pointer |
    +-------+---------+
[0] | "Tom" |  None   |
[1] |  ""   |  None   |
[2] | "Ted" |    0    |
[3] | "Sue" |    4    |
[4] | "Ann" |  None   |
    +-------+---------+

BoyListPointer
   |
   |
+-------+-----+   +-------+------+
| "Ted" |  o----->| "Tom" | None |
+-------+-----+   +-------+------+

GirlListPointer
   |
   |
+-------+-----+   +-------+------+
| "Sue" |  o----->| "Ann" | None |
+-------+-----+   +-------+------+

FreeListPointer
     |
     |
+----+------+
| "" | None |
+----+------+
```

This provides an analogy for how linked lists are stored within the computer's memory. Multiple linked lists may not necessarily occupy contiguous blocks and their memory cells may be intermingled with one another.

Doubly Linked List & Circular Linked List [Not in Syllabus]

In a **doubly linked list**, each node has two pointers: one to the previous node and one to the next node. This allows us to traverse the linked list in two directions. An example of a doubly linked list is the list of webpages accessed when you are browsing the Internet. You can traverse the list using the back and forward buttons on your browser.

In a **circular linked list**, the pointer of the last node points to the first node. An example of a circular linked list is a music or video playlist that loops back to the first item after the last one has finished playing.

Doubly linked list and circular linked list are not in the syllabus, but once you have understood the concept of linked list well, they should not be difficult to implement.

Appendix

You may want to watch the following video to help you understand linked lists better.

https://www.youtube.com/watch?v=_jQhALI4ujg (until 6:25 for the syllabus)

2020 JC1 H2 Computing 9569

19. Pseudocode

In computing, solutions to problems are often written in **pseudocode**. This resembles a programming language, but does not follow the syntax of any one particular language, so that users of different languages can all read it.

Common Constructs

The following table shows some common constructs used in programming, expressed both in Python and pseudocode.

	Pseudocode	**Python**
Declaration	`DECLARE A : INTEGER`	Python does not have variable declaration
Assignment	`A ← 34`	`A = 34`
Changing a value	`B ← B + 1`	`B = B + 1`

If/then/else	`IF A > B`
	`  THEN ...`
	`  ELSE ...`
	`ENDIF`
		`if A > B:`
		`  ...`
		`else:`
		`  ...`

Repetition (while loop)	`REPEAT`
	`  ...`
	`UNTIL A > B`
	
	Alternatively,
	`WHILE A <= B`
	`  ...`
	`ENDWHILE`
		`while A <= B:`
		`  ...`

Iteration (for loop)	`FOR N ← 0 TO 10`
	`  ...`
	`ENDFOR`
		`for N in range(11):`
		`  ...`

Input	`INPUT "Prompt:" A`	`a = input("Prompt:")`
Output	`OUTPUT "Message" B`	`print("Message")` `print(B)`
Comment	`// Comment`	`# Comment`

The following table shows relational operators and mathematical functions.

	Pseudocode	**Python**
is equal to	=	=

	Pseudocode	Python
is less than	<	<
is greater than	>	>
is less than or equal to	<=	<=
is greater than or equal to	>=	>=
is not equal to	<>	!=
addition	+	+
subtraction	-	-
multiplication	*	*
division	/	/
exponentiation	^	**
integer division (quotient)	DIV	//
modulus (remainder)	MOD	%

Exercise 1

(a) What does the following algorithm written in pseudocode do?
(b) Try to write the Python equivalent.

```
DECLARE n: INTEGER       // While variable declaration is not
compulsory,
DECLARE Total: INTEGER   // it is a good practice to do so, so
that other people
DECLARE Count: INTEGER   // reading your code know what data
type it is.

n ← 0

WHILE n <= 0
    INPUT n
ENDWHILE

Total ← 0
Count ← 0

REPEAT
    INPUT Number
    Total ← Total + Number
    Count ← Count + 1
UNTIL Count = n

OUTPUT Total/Count
```

In [ ]:
```python
# Find average of n numbers
# Type your code here
n = 0
while n <= 0:
    n = int(input("Enter number of numbers to input: "))

total = 0
count = 0

while count != n:
    number = int(input(f"Enter number {count + 1}: "))
    total += number
    count += 1

print(total/count)
```

Exercise 2

(a) What does the following algorithm written in pseudocode do?

(b) Try to write the Python equivalent.

```
INPUT a
INPUT b
INPUT c

d ← (b*b) - (4*a*c)

IF d < 0
    THEN
        OUTPUT "There are no real roots."
    ELSE
        SquareRoot ← SQRT(d)                    // For common
mathematical functions, we can use abbreviations
        Root1 ← (-b + SquareRoot) / (2*a)
        Root2 ← (-b - SquareRoot) / (2*a)
        OUTPUT Root1 and Root2
ENDIF
```

In [ ]:
```python
# Find real roots of quadratic equation
# Type your code here
a = int(input())
b = int(input())
c = int(input())
d = b*b - 4*a*c

if d < 0:
    print("There are no real roots.")
else:
    SquareRoot = d**0.5
    Root1 = (-b + SquareRoot) / (2*a)
    Root2 = (-b - SquareRoot) / (2*a)
    print(Root1, Root2)
```

Exercise 3

(a) What does the following algorithm written in pseudocode do?

(b) Try to write the Python equivalent.

```
INPUT Value1
INPUT Value2
```

```
Temp ← Value1
Value1 ← Value2
Value2 ← Temp
```

In [ ]:
```
# Swap two values with temporary variable
# Type your code here
Value1 = input("Input value 1: ")
Value2 = input("Input value 2: ")

Temp = Value1
Value1 = Value2
Value2 = Temp
```

Exercise 4

(a) What does the following algorithm written in pseudocode do?
(b) Try to write the Python equivalent.

```
INPUT Value1
INPUT Value2

Value1 ← Value1 + Value2
Value2 ← Value1 – Value2
Value1 ← Value1 – Value2
```

In [2]:
```
#Swap two values without temporary variable
# Type your code here
Value1 = int(input("Input value 1: "))
Value2 = int(input("Input value 2: "))

Value1 = Value1 + Value2
Value2 = Value1 - Value2
Value1 = Value1 - Value2
```

```
Input value 1: 1
Input value 2: 2
```

Exercise 5

(a) What does the following algorithm written in pseudocode do?
(b) Try to write the Python equivalent.

What does the following algorithm written in pseudocode do? Try to write a Python equivalent.

```
DECLARE List1: ARRAY[1:3] of INTEGER     // This is a 1D array
of integers of size 3.
DECLARE List2: ARRAY[1:3] of INTEGER     // This is another 1D
array of integers of size 3.

FOR Index ← 1 TO 3
    List1[Index] ← Index*2
ENDFOR

FOR Index ← 1 TO 3
    List2[Index] ← Index^2
ENDFOR
```

In [6]:
```
# Get a list of consecutive and square numbers
```

```python
# Type your code here
List1 = []
List2 = []

for i in range(1, 4):
    List1.append(i*2)

for i in range(1, 4):
    List2.append(i**2)
```

For 2D arrays,

e.g. declaration of a 2D array of strings of size 3x5:

```
DECLARE List: ARRAY[0:2, 0:4] of STRING
```

e.g. accessing and outputting an element in a 2D array:

```
OUTPUT List[1, 2]   // This is the equivalent of List[1][2] in
Python
```

String

The following table shows common string functions and methods.

	Pseudocode	Python
Returns the start position of `str2` in `str1`, or -1 if `str2` is not in `str1`	`LOCATE(str1, str2)`	`str1.find(str2)`
Returns the first `n` characters of `str`	`LEFT(str, n)`	`str[0:n]`
Returns (as a string) the next `n` characters of `str`, starting with the `m` th character	`MID(str, m, n)`	`str[m:m+n]`
Returns the last `n` characters of `str`	`RIGHT(str, n)`	`str[-n:]`
Returns the number of characters in `str`	`LENGTH(str)`	`len(str)`
Concatenates `str1` and `str2`	`str1 & str2` Alternatively, `CONCAT(str1, str2)`	`str1 + str2`

For instance, suppose we want to come up with a procedure to draw a pyramid of characters as shown below.

```
   A
  AAA
 AAAAA
AAAAAAA
```

One solution is:

```
INPUT character                          // the character to be
used in the pyramid

REPEAT
    INPUT MaxNumberOfCharacters          // the number of
characters in the bottom row (must be odd)
UNTIL MaxNumberOfCharacters MOD 2 = 1

NumberOfSpaces ← (MaxNumberOfCharacters - 1)/2
NumberOfCharacters ← 1

REPEAT
    FOR i ← 1 TO NumberOfSpaces
        OUTPUT < ' Space' >
    ENDFOR

    FOR i ← 1 TO NumberOfCharacters
        OUTPUT character
    ENDFOR

    OUTPUT Newline      // This goes to the next line. It is
like pressing "enter" or using "\n" in Python.

    NumberOfSpaces ← NumberOfSpaces - 1
    NumberOfCharacters ← NumberOfCharacters + 2
UNTIL NumberOfCharacters > MaxNumberOfCharacters
```

Exercise 6

(a) What does the following algorithm written in pseudocode do?

(b) Try to write the Python equivalent.

```
INPUT MyName
LengthOfName ← LENGTH(MyName)
PositionOfSpace ← LOCATE(MyName, '< Space >')

FirstWord ← LEFT(MyName, PositionOfSpace)
RestOfName ← RIGHT(MyName, (LengthOfName - PositionOfSpace - 1))

OUTPUT RestOfName & '< Space >' & FirstWord
```

In [12]:
```
# Flips name
# Type your code here
MyName = input()
LengthOfName = len(MyName)
PositionOfSpace = MyName.find(" ")
FirstWord = MyName[:PositionOfSpace]
RestOfName = MyName[-(LengthOfName - PositionOfSpace - 1):]

print(RestOfName + ' ' + FirstWord)
```

```
1
1
```

Exercise 7

The following algorithm inputs seven numbers into a list.

```
DECLARE MyList: ARRAY[0:6] of INTEGER
```

```
FOR Index ← 0 TO 6
    INPUT MyList[Index]
ENDFOR
```

What does the following algorithm written in pseudocode do?

```
MaxIndex ← 6
INPUT SearchValue
Found ← FALSE
Index ← 0

REPEAT
    IF MyList[Index] = SearchValue
        THEN
            Found ← TRUE
    ENDIF
    Index ← Index + 1
UNTIL FOUND = TRUE OR Index > MaxIndex

IF Found = TRUE
    THEN
        OUTPUT "Value found at location:" Index
    ELSE
        OUTPUT "Value not found"
ENDIF
```

Procedures and Functions

A **procedure** refers to a group of steps carried out in a fixed order. In Python, procedures and functions are not really distinguished. You may think of a procedure as a function that does not return anything.

We can re-do the pyramid of characters using procedures, calling them one at a time.

```
CALL SetValues
REPEAT
    CALL OutputSpaces
    CALL OutputCharacters
    CALL AdjustValuesForNextRow
UNTIL NumberOfCharacters > MaxNumberOfCharacters
```

Now, we need to define the procedures that we need. These are:

```
PROCEDURE SetValues
    INPUT Character
    CALL InputMaxNumberOfCharacters // We called yet another
procedure which we need to define.
    NumberOfSpaces ← (MaxNumberOfCharacters - 1)/2
    NumberOfCharacters ← 1
ENDPROCEDURE

PROCEDURE InputMaxNumberOfCharacters
    REPEAT
        INPUT MaxNumberOfCharacters
    UNTIL MaxNumberOfCharacters MOD 2 = 1
ENDPROCEDURE
```

```
PROCEDURE OutputSpaces
    FOR Count ← 1 TO NumberOfSpaces
        OUTPUT '< Space >'
    ENDFOR
ENDPROCEDURE

PROCEDURE OutputCharacters
    FOR Count ← 1 TO NumberOfCharacters
        OUTPUT Character
    ENDFOR
    OUTPUT Newline
ENDPROCEDURE

PROCEDURE AdjustValuesForNextRow
    NumberOfSpaces ← NumberOfSpaces - 1
    NumberOfCharacters ← NumberOfCharacters + 2
ENDPROCEDURE
```

When we write a function in pseudocode, we need to specify the input (if any), as well as what it must return as shown in the example below.

Exercise 8

(a) What does the following algorithm written in pseudocode do?
(b) Try to write the Python equivalent.

```
FUNCTION InputOddNumber() RETURNS INTEGER
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    RETURN Number
ENDFUNCTION
```

In [16]:
```python
# Input odd number and returns it
# Type your code here
def InputOddNumber():
    Number = 0
    while Number % 2 != 1:
        Number = int(input("Enter an odd number: "))
    return Number
```

When a procedure or function requires values from the main program, these are supplied as arguments. When we define a function, we put them in a **parameter list**. The list specifies the parameters needed by the function, as well as their data types. The parameters are said to be **passed** to the procedure or function.

Refer to the example below.

```
FUNCTION SumAP(FirstTerm: INTEGER, LastTerm: INTEGER) RETURNS
INTEGER
DECLARE Sum, CurrTerm : INTEGER
    Sum ← 0
    FOR CurrTerm ← FirstTerm TO LastTerm
        Sum ← Sum + CurrTerm
    ENDFOR
    RETURN Sum
ENDFUNCTION
```

In [22]:
```
def sumAP(FirstTerm, LastTerm):
    Sum = 0
    for CurrTerm in range(FirstTerm, LastTerm+1):
        Sum = Sum + CurrTerm
    return Sum
```

A parameter may be passed **by value**, as we saw in the function above. When the argument is called, a copy of the value is passed into the function. The value of the variable in the main program is not affected by what happens inside the function.

Returning to the earlier example where we printed a pyramid of characters, the procedure `OutputCharacters` passes two parameters by value.

Alternatively, a parameter may be passed **by reference**. When the argument is called, a pointer to the memory location is passed into the function or procedure. Changes which are applied to the variable's contents will be effective outside the procedure, i.e. in the main program. The procedure `AdjustValuesForNextRow` passes two variables by reference.

We now make this clear in the pseudocode:

```
PROCEDURE OutputCharacters(BYVAL NumberOfCharacters : INTEGER,
BYVAL Character: CHAR)
DECLARE Count : INTEGER
    FOR Count ← 1 TO NumberOfCharacters
        OUTPUT Character
    ENDFOR
    OUTPUT Newline
ENDPROCEDURE

PROCEDURE AdjustValuesForNextRow(BYREF Spaces : INTEGER, BYREF
Characters : INTEGER)
    Spaces ← Spaces - 1
    Characters ← Characters + 2
ENDPROCEDURE
```

We also need to amend the main program:

```
CALL SetValues
REPEAT
    CALL OutputSpaces
    CALL OutputCharacters
    CALL AdjustValuesForNextRow(NumberOfSpaces,
NumberOfCharacters)
UNTIL NumberOfCharacters > MaxNumberOfCharacters
```

As Python does not distinguish between these two methods of passing parameters in its code, do be careful when using the same variable name repeatedly in Python.

In [ ]:
```
def OutputCharacters(NumberOfCharacters, Character):
    for Count in range(NumberOfCharacters):
        print(Character, end='')
    print()

OutputCharacters(5, 'A')
```

In [ ]:
```
def AdjustValuesForNextRow(Spaces, Characters):
    Spaces = Spaces - 1
    Characters = Characters + 1
    return Spaces, Characters

NumberOfSpaces = int(input())
NumberOfCharacters = int(input())
NumberOfSpaces, NumberOfCharacters = AdjustValuesForNextRow(NumberOfSpaces, N
print(NumberOfSpaces)
print(NumberOfCharacters)
```

Finally, the following shows the recursive pseudocode for calculating a function that calculates the factorial of an input integer.

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0
        THEN
            Result ← 1
        ELSE
            Result ← n * Factorial(n-1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

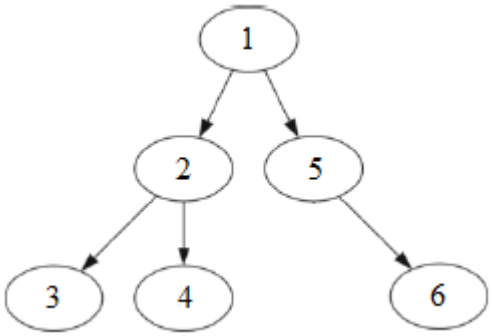
2020 JC1 H2 Computing 9569

20. Tree

Now that we have understood the idea of recursion, we shall move on to another important data structure that is commonly used in many areas of Computer Science, including but not limited to database, graphics, networking and operating systems.

Tree is a non-linear, hierarchical data structure with a root, branches and leaves, just like its botanical counterpart. The difference is that a tree data structure has its root at the top and its leaves below.

Here is an example of a tree.



Let us understand the vocabulary used in relation to trees.

Node
Represented by an oval, each node stores a value.
In the example above, each node contains an integer.

Edge
Represented by an arrow, each edge connects two nodes to show the relationship between them.

Root
It is the only node that has no incoming edge.
In the example above, the node storing the integer 1 is the root.

Parent, Child & Sibling
A parent node is connected to the child node through an edge. The children of the same parent are siblings.
In the example above, the parent node storing the integer 1 has two children nodes storing the integers 2 and 5 respectively.

Subtree (Branch)
A subtree is a set of nodes and edges comprising a parent and all the descendants of that parent.

Leaf
A leaf node is a node that has no children.

Path
A path is an ordered list of nodes that are connected by edges.
In the example above, 1 -> 2 and 1 -> 5 -> 6 are two examples of the available paths.

Level
The level of a node refers to its depth in the tree. By definition, the level of the root node is 0.

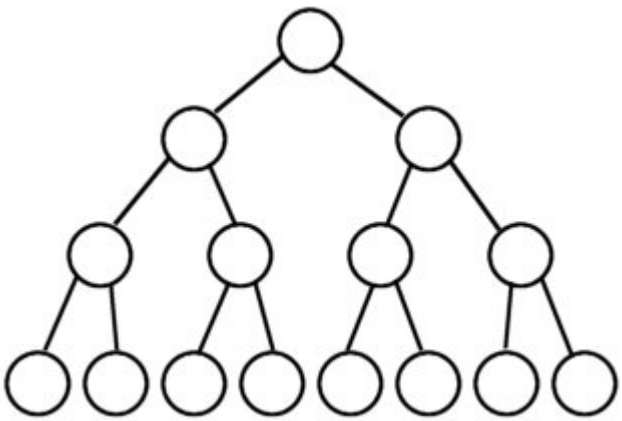
Height
The height of a tree tells us how deep the entire hierarchy is.
In the example above, the height is 3.

Size
The size of a tree is the total number of nodes present.
In the example above, the size is 6.

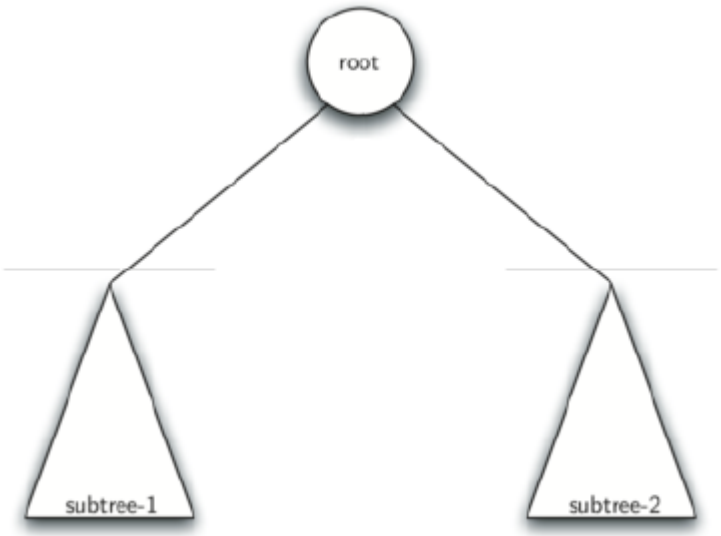
Binary Tree

Binary tree is a tree where each node has at most two children (commonly referred to as the left child and the right child).

In a perfect binary tree of a given height *h* where each node has two children except for the leaf nodes, the size is $2^h - 1$.



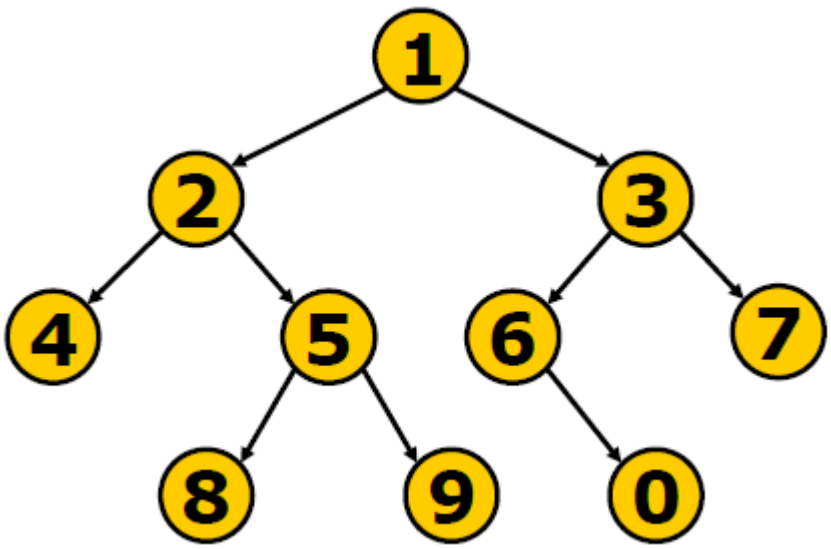
Besides the root node, every node except for the leaf nodes can be seen as a root node of a smaller tree. Hence, a tree can be an empty tree or consists of a root and zero or more subtrees. The idea of recursion should come to your mind by now.



Tree Traversal

We shall look at three different ways of traversing a tree, which form the basis of depth-first search.

To illustrate, we shall use the following binary tree.



Pre-Order Traversal

In this mode of traversal, we traverse the root first, followed by the left subtree and finally the right subtree.

Result: 1 -> 2 -> 4 -> 5 -> 8 -> 9 -> 3 -> 6 -> 0 -> 7

In-Order Traversal

In this mode of traversal, we traverse the left subtree first, followed by the root and finally the right subtree.

Result: 4 -> 2 -> 8 -> 5 -> 9 -> 1 -> 6 -> 0 -> 3 -> 7

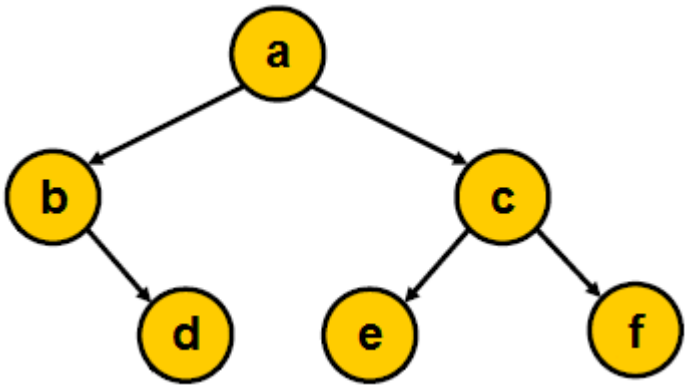
Post-Order Traversal

In this mode of traversal, we traverse the left subtree first, followed by the right subtree and finally the root.

Result: 4 -> 8 -> 9 -> 5 -> 2 -> 0 -> 6 -> 7 -> 3 -> 1

Exercise

Refer to the following binary tree.

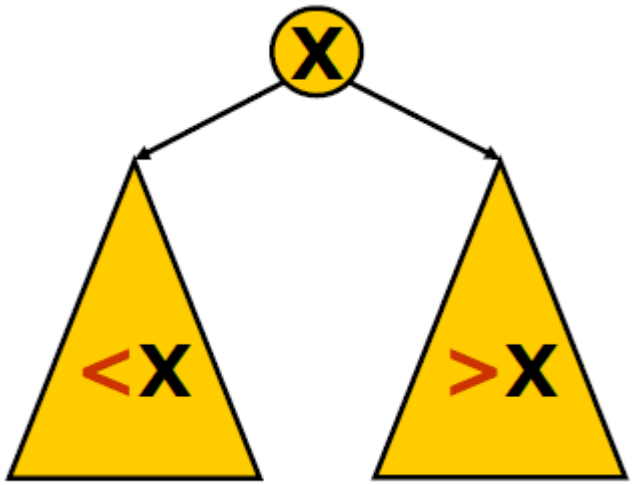


Traverse the tree using the three methods described above.

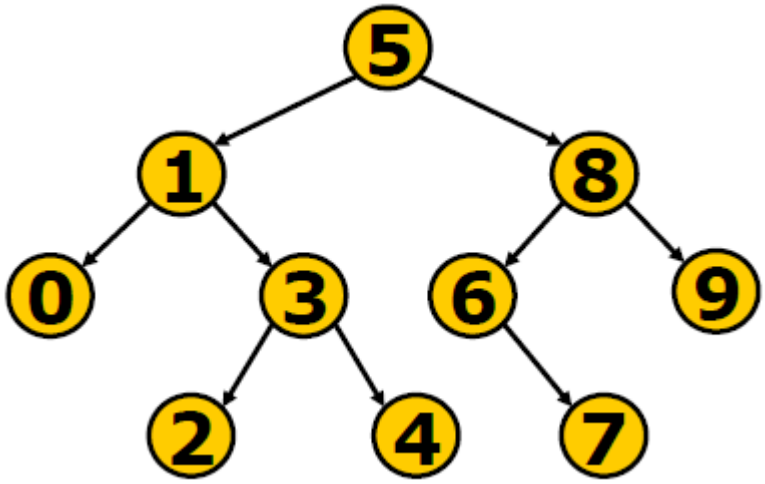
In [ ]:
```
# Type your answer below.

# Pre-order  : a, b, d, c, e, f

# In-order   : b, d, a, e, c, f

# Post-order : d, b, e, f, c, a
```

Binary Search Tree (BST)

We are interested in a special type of binary tree called **binary search tree**. In this tree, all values smaller than the root value are stored in the left subtree, while all values larger than the root value are stored in the right subtree.



Below is an example of a binary search tree.



Notice that in-order traversal of a binary search tree gives values sorted in increasing order, which is from 0 to 9.

Implementing BST

Node Class

First of all, we need to define the *Node* class, the objects of which shall be the building blocks for our BST.

In [ ]:
```
class Node:
    def __init__(self, data):
        self.data = data    # primitive data type, e.g. integer, string
        self.left = None    # this shall point to the left child node in the
        self.right = None   # this shall point to the right child node in the
```

BST Class

The *BST* class should have only one property, which is its root. When creating a new BST, it should be empty.

Some of the methods of BST that we need to be familiar with include:

1. Traversal

Print the data in the BST for pre-order, in-order and post-order traversals.

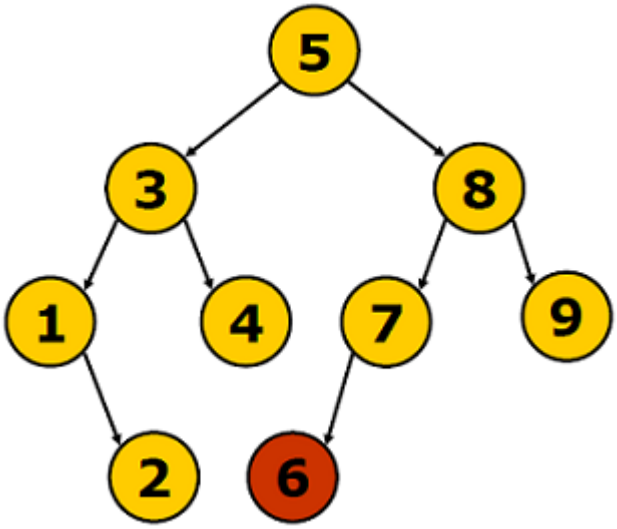
2. Search

Search for a node with the given data in the BST. It returns True when the data can be found, and False otherwise.

3. Insert

Insert a node with the given data into the BST at the correct position. This is done by tracing the tree from the root node, doing comparison with the value of the node before deciding which path to be taken.

The example below shows where a new node with a value of 6 is inserted into the binary search tree.



4. Delete

Delete a node with the given data from the BST and possibly rearrange the tree.

In the syllabus, the focus is on the conceptual understanding of how nodes are deleted from binary search tree. Writing algorithms and programs to delete nodes from a BST is excluded from the syllabus.

Example:

- If the node to be deleted is a leaf node (i.e. nodes 2, 4 or 6), it is quite a straightforward process as there is no need to shift any of the remaining nodes.
- If the node to be deleted has one child (e.g. node 1), node 2 will be the new left child of node 3.
- If the node to be deleted has two children (e.g. node 5), there are two possible scenarios:
    - move the smallest node in the right subtree to the position of the deleted node. i.e. node 6 becomes the root node of the BST.
    - move the largest node in the left subtree to the position of the deleted node. i.e. node 4 becomes the root node of the BST.

Define the *BST* class.

In [ ]:
```
class BST:
    # The BST is empty when initialised
    def __init__(self):
        self.root = None

    # Inserts a new node with the given data to the BST
    def insert(self, data):
        def insert_helper(cur, data):
            if data < cur.data:
                # If there exists a left child, traverse to the left child vi
                if cur.left:
                    insert_helper(cur.left, data)
                # Else, create a new node as the left child
                else:
```

```
                cur.left = Node(data)
            else:
                # If there exists a right child, traverse to the right child
                if cur.right:
                    insert_helper(cur.right, data)
                # Else, create a new node as the left child
                else:
                    cur.right = Node(data)

        # If the tree is empty, create a new node as the root
        if self.root == None:
            self.root = Node(data)
        # Else, start to traverse the tree by calling the helper function
        else:
            insert_helper(self.root, data)

    # Searches for a node with the given data in the BST
    # Returns True when found, and False otherwise
    def search(self, data):
        def search_helper(cur, data):
            if cur == None:
                return False
            elif data == cur.data:
                return True
            elif data < cur.data:
                return search_helper(cur.left, data)
            else:
                return search_helper(cur.right, data)

        return search_helper(self.root, data)

    # Prints out the result of pre-order traversal of the BST
    def pre_order(self):
        def pre_order_helper(cur):
            print(cur.data, end=' ')
            if cur.left:
                pre_order_helper(cur.left)
            if cur.right:
                pre_order_helper(cur.right)

        # If the tree is empty
        if self.root == None:
            print("Tree is empty!")
        else:
            pre_order_helper(self.root)

    # Prints out the result of in-order traversal of the BST
    def in_order(self):
        def in_order_helper(cur):
            if cur.left:
                in_order_helper(cur.left)
            print(cur.data, end=' ')
            if cur.right:
                in_order_helper(cur.right)

        # If the tree is empty
        if self.root == None:
            print("Tree is empty!")
        else:
            in_order_helper(self.root)

    # Prints out the result of post-order traversal of the BST
    def post_order(self):
        def post_order_helper(cur):
            if cur.left:
```

```
                post_order_helper(cur.left)
            if cur.right:
                post_order_helper(cur.right)
            print(cur.data, end=' ')

        # If the tree is empty
        if self.root == None:
            print("Tree is empty!")
        else:
            post_order_helper(self.root)

    # Extension:
    # Instead of printing out the result of a traversal,
    # we can also put the elements in a list and return it
    # e.g. for post-order traversal:
    def post_order_list(self):
        def post_order_list_helper(cur):
            if cur.left:
                post_order_list_helper(cur.left)
            if cur.right:
                post_order_list_helper(cur.right)
            result.append(cur.data)

        result = []

        if self.root:
            post_order_list_helper(self.root)

        return result

# Test case
bst1 = BST()
bst1.insert(40)
bst1.insert(20)
bst1.insert(60)
bst1.insert(10)
bst1.insert(30)
bst1.insert(50)
```

Setting Up a Binary Tree Inside an Array

Similar to the linked list, it is also possible to use an array representation for a binary tree. In this case, we first set up a free list of unused memory cells. Each node is a list of *three* elements: the data, the left pointer, and the right pointer. The pointers point to the respective children of the node. In the free list, for convenience, the nodes are put in order so that each node is the left child of the previous node.

In [ ]:

```
class Array:

    # The array has indices 0 to n-1 (inclusive).
    # Each entry in the array is a node, which will be a list of three elemen

    def __init__(self, n):
        self.Array = []
        for i in range(n-1):
            node = ["", i+1, None]

            # node[0] the data, which we initialise using an empty string
            # node[1] is the left pointer
            # node[2] is the right pointer

            self.Array.append(node)
```

```
        self.Array.append(["", None, None])

        self.FreeListPointer = 0 # head pointer of free list
```

Our array now looks like this (where we use `n=5`):

```
FreeListPointer = 0

Array:
    +------+--------------+---------------+
    | Data | Left Pointer | Right Pointer |
    +------+--------------+---------------+
[0] |  ""  |       1      |      None     |
[1] |  ""  |       2      |      None     |
[2] |  ""  |       3      |      None     |
[3] |  ""  |       4      |      None     |
[4] |  ""  |     None     |      None     |
    +------+--------------+---------------+
```

Suppose we want to create a binary tree called `MyTree`. Initially it is empty.

```
MyTreePointer = None
FreeListPointer = 0

Array:
    +------+--------------+---------------+
    | Data | Left Pointer | Right Pointer |
    +------+--------------+---------------+
[0] |  ""  |       1      |      None     |
[1] |  ""  |       2      |      None     |
[2] |  ""  |       3      |      None     |
[3] |  ""  |       4      |      None     |
[4] |  ""  |     None     |      None     |
    +------+--------------+---------------+
```

We insert `"Tom"` into `MyTree`:

```
MyTreePointer = 0
FreeListPointer = 1


Array:
    +-------+--------------+---------------+
    | Data  | Left Pointer | Right Pointer |
    +-------+--------------+---------------+
[0] | "Tom" |     None     |      None     |
[1] |  ""   |       2      |      None     |
[2] |  ""   |       3      |      None     |
[3] |  ""   |       4      |      None     |
[4] |  ""   |     None     |      None     |
    +-------+--------------+---------------+
</pre>
We insert "Sue" as a left child of "Tom".


MyTreePointer = 0
FreeListPointer = 2
```

Array:

	Data	Left Pointer	Right Pointer
[0]	"Tom"	1	None
[1]	"Sue"	None	None
[2]	""	3	None
[3]	""	4	None
[4]	""	None	None

</pre>
We insert "Bob" as a right child of "Tom".

MyTreePointer = 0
FreeListPointer = 3

Array:

	Data	Left Pointer	Right Pointer
[0]	"Tom"	1	2
[1]	"Sue"	None	None
[2]	"Bob"	None	None
[3]	""	4	None
[4]	""	None	None

</pre>
We delete "Tom" and make "Bob" the new parent of "Sue":

MyTreePointer = 2
FreeListPointer = 0

Array:

	Data	Left Pointer	Right Pointer
[0]	""	3	None
[1]	"Sue"	None	None
[2]	"Bob"	1	None
[3]	""	4	None
[4]	""	None	None

</pre>
In this way, the abstract tree structure is maintained by use of pointers, even though the order in which the data appears in the tree may be completely different from the order in which it appears in the array.

2020 JC1 H2 Computing 9569

21. Time Complexity

Even though computers can carry out millions (maybe billions) of intructions per second, efficiency is still a major factor to consider when writing programs. Suppose that Program A runs ten times faster than Program B. If the input is small, it may be a difference of between 0.1 seconds and 0.01 seconds, which might not be noticeable to us. However, if the input is way larger, the difference between one hour and ten hours is certainly noticeable.

Example 1

Consider the following two methods to find the sum of an arithmetic progression:

Method 1

```
INPUT FirstTerm
INPUT CommonDiff
INPUT NoOfTerms

Sum ← 0
CurrTerm ← FirstTerm

FOR Count ← 1 TO NoOfTerms
    Sum ← Sum + CurrTerm
    CurrTerm ← CurrTerm + CommonDiff
ENDFOR

OUTPUT Sum
```

Method 2

```
INPUT FirstTerm
INPUT CommonDiff
INPUT NoOfTerms

Sum ← (NoOfTerms / 2)*(2*FirstTerm + (NoOfTerms - 1)*CommonDiff)

OUTPUT Sum
```

How many steps does each method take?

Big-O Notation

The **Big-O notation** is commonly used as an indication of the number of steps (and hence the time) taken for the program to run.

Example 2

Consider the following method to determine whether a number *n* is prime.

Method 1

```
INPUT n
Result ← TRUE

IF n = 1
    THEN
        Result ← FALSE
    ELSE
        IF n > 2         // What happens when n is 2?
            THEN
                FOR x ← 2 TO n-1
                    Remainder = n MOD x
                    IF Remainder = 0
                        THEN
                            Result ← FALSE
                    ENDIF
                ENDFOR
        ENDIF
ENDIF

OUTPUT Result
```

The number of steps is determined by the number of times the FOR loop happens. Within the FOR loop, the number of steps is (roughly) constant. Since we go through the loop *n*-2 times, the total number of steps is (roughly) a multiple of *n*-2. For large values of *n*, the total number of steps is approximately a multiple of *n*, therefore, we say that this program has a time complexity of O(*n*).

It is possible to improve the program. In the way it is written above, the program has to perform the loop *n*-1 times regardless of the value of Result. We could break the loop once Result becomes FALSE.

Method 2

```
INPUT n
Result ← TRUE

IF n = 1
    THEN
        Result ← FALSE
    ELSE
        x ← 2
        REPEAT
            Remainder = n MOD x
            IF Remainder = 0
                THEN
                    Result ← FALSE
            ENDIF
            x ← x+1
        UNTIL x > n-1 OR Result = FALSE
ENDIF

OUTPUT Result
```

In this case, the loop breaks once we find a value of x that is a factor of n (and between 1 and n, exclusive). However, in the worst-case scenario, which is when n is actually prime, we still need to go through the loop n-2 times. Therefore, in the worst-case scenario, the number of steps is still (roughly) a multiple of n, so the time complexity is still O(n).

Is it possible to improve the program beyond this? If n has a factor between 1 and n, then it must have a factor between 1 and $\sqrt{n}$ (why?). Therefore, we only need to check for factors up to $\sqrt{n}$.

Method 3

```
INPUT n
Result ← TRUE

IF n = 1
    THEN
        Result ← FALSE
    ELSE
        SQUAREROOT = SQRT(n)
        x ← 2
        REPEAT
            Remainder = n MOD x
            IF Remainder = 0
                THEN
                    Result ← FALSE
            ENDIF
            x ← x+1
        UNTIL x > SQUAREROOT OR Result = FALSE
ENDIF

OUTPUT Result
```

In this case, even in the worst case scenario, we go through the loop at most $\sqrt{n}$ -1 times. The time complexity of this is O($\sqrt{n}$).

Returning to Example 1, where we tried to find the sum of an arithmetic progression, we notice that Method 1 goes through the loop n times, where n is the number of terms in the progression, and therefore it is O(n). In Method 2, the number of steps is the same regardless of the values of the inputs, and so it as a constant number of steps, which we indicate is O(1).

Example 3

Suppose we want to check whether two lists have any elements in common. The code in Python would look something like this.

In [ ]:
```
def checkcommon(lis1, lis2):
    for item in lis1:
        if item in lis2:
            return True
    return False
```

The compactness of the code in Python obscures the fact that we actually need to iterate through both lists. In pseudocode, the same algorithm would look like this:

```
INPUT List1
INPUT List2

Length1 ← LENGTH(List1)
Length2 ← LENGTH(List2)

Result ← FALSE

Index1 ← 0
Index2 ← 0

REPEAT
    item1 ← List1[Index1]
    REPEAT
        item2 ← List1[Index2]
        IF item1 = item2
            THEN
                Result ← TRUE
        ENDIF
        Index2 ← Index2 + 1
    UNTIL Result = TRUE OR Index2 >= Length2
UNTIL Result = TRUE OR Index1 >= Length1

OUTPUT Result
```

Notice that we have a nested loop. In the worse case scenario (where the two lists do not have any common elements), we have to go through the outside loop for each element in List1, and through the inside loop for each element in List2. Therefore, the total number of steps is (roughly) a multiple of *mn*, where *m* and *n* are the number of elements in List1 and List2 respectively. Therefore, this program has a time complexity of O(*mn*).

Example 4

Suppose we want to calculate the *n*th power of *x*, where *n* is a positive integer. In Python, this would simply be `x\*\*n` , but suppose we do not have the luxury of using `\*\*` , and have to multiply repeatedly. The straightforward way to do it is

```
INPUT x
INPUT n

Result ← 1

FOR count ← 1 TO n
    Result ← Result*x
ENDFOR

OUTPUT Result
```

which has a time complexity of O(*n*). Is there a better way to do it? Surprisingly, the answer is yes.

We first define a function to convert *n* into binary notation.

```
FUNCTION Dec_to_Bin(n : INTEGER) RETURNS STRING
    Result ← '' // There is no standard way to express the empty
```

```
string in pseudocode. Sometimes ⌀ is used as well.
    Temp ← n

    REPEAT
        IF Temp MOD 2 = 0
            THEN
                Result ← Result & '0'
            ELSE
                Result ← Result & '1'
        ENDIF
        Temp ← Temp DIV 2
    UNTIL Temp = 0

    RETURN Result
ENDFUNCTION
```

What is the time complexity of `Dec_to_Bin`? Go through the following algorithm for finding x^n to see why it works.

```
INPUT x
INPUT n

Str ← Dec_to_Bin(n)
Temp ← x
Result ← 1
Length = LENGTH(Str) // Can you find what is Length in terms of
n?

FOR i ← 1 TO Length
    Temp ← Temp*Temp
    IF MID(Str, Length-i, 1) = 1
        THEN
            Result ← Result*Temp
    ENDIF
ENDFOR

OUTPUT Result
```

What is the total time complexity? We called the function `Dec_to_Bin` when we were finding `Str` so the time complexity of `Dec_to_Bin` plays a role here. In addition, we also went through the loop `Length` times. Therefore, the time complexity of this algorithm is __.

Mathematical Formalities

Suppose we have two functions f(n) and g(n), such that there exist constants C and n_0 such that f(n) ≤ Cg(n) for all values of $n > n_0$. Then we say that f(n) = O(g(n)).

For instance, if f(n) = 3n + 5 and g(n) = n^2, then f(n) = O(g(n)) since f(n) ≤ 1g(n) for all $n > 4$ (you can prove this with graphically or with algebra). However, g(n) ≠ O(f(n)) since for every value of C, we will have g(n) ≥ Cf(n) once n is large enough.

Notice that if f(n) and g(n) are both polynomials of the same degree, then f(n) = O(g(n)) and g(n) = O(f(n)). We say that f(n) and g(n) are of the **same order**.

On the other hand, suppose $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$ (as in the example above). If Algorithm A takes $f(n)$ and Algorithm B takes $g(n)$ steps, then we say that Algorithm A is **more efficient** than Algorithm B, since the Algorithm A will take less time to compute the result once n is sufficiently large. Notice that this doesn't mean Algorithm A takes less time for *all* values of n. For instance, if if $f(n) = 100n$ and $g(n) = 0.01n^2$, $f(n)$ will be larger than $g(n)$ for small values of n (in fact, up to 10,000). The point is that $g(n)$ will eventually catch up and overtake $f(n)$.

We can therefore arrange functions in order of efficiency. A rough guide is

More efficient $O(1)$, $O(\log n)$, $O(\sqrt{n})$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, ..., $O(2^n)$, $O(3^n)$, ..., $O(n!)$, ... **Less efficient**

In general, we say that an algorithm is efficient if it is $O(n^k)$ or faster. These algorithms are known as **polynomial-time** algorithms. Algorithms which are slower (e.g. $O(2^n)$) are known as non-polynomial time algorithms. Currently, there is intense research going on to prove that there are some tasks which cannot be performed in polynomial time.

2020 JC1 H2 Computing 9569

22. Sorting Algorithms

One of the basic tasks a computer scientist needs to be able to do is **sort**, that is, to arrange the elements of a list in a non-decreasing order. (The word 'non-decreasing' is used instead of 'increasing' because of the possibility that the list may contain two or more copies of the same element.) In other words, the list should be rearranged so that each element is less than or equal to the element that comes after it.

Selection Sort

This is the simplest, and perhaps the most natural, sorting algorithm. However, it turns out to be slow compared to other algorithms that we will learn. We go through the entire list, look for the smallest element, and swap it with the element at index 0. This constitutes one **pass**. In the next pass, we go through the list from index 1 to the end, look for the smallest element, and swap it with the element at index 1. We repeat the process from index 2 to the end, and so on. Once the correct element is placed in the second last slot, the last slot will automatically have the largest element.

Example

```
[25, 34, 98,  7, 41, 19,  5] // START:
[25, 34, 98,  7, 41, 19,  5] // smallest element is 5
                         ^
[ 5, 34, 98,  7, 41, 19, 25] // swap it with list[0]
[ 5, 34, 98,  7, 41, 19, 25] // smallest element in list[1:] is
7
              ^
[ 5,  7, 98, 34, 41, 19, 25] // swap it with list[1]
[ 5,  7, 98, 34, 41, 19, 25] // smallest element in list[2:] is
19
                     ^
[ 5,  7, 19, 34, 41, 98, 25] // swap it with list[2]
[ 5,  7, 19, 34, 41, 98, 25] // smallest element in list[3:] is
25
                        ^
[ 5,  7, 19, 25, 41, 98, 34] // swap it with list[3]
[ 5,  7, 19, 25, 41, 98, 34] // smallest element in list[4:] is
34
                        ^
[ 5,  7, 19, 25, 34, 98, 41] // swap it with list[4]
[ 5,  7, 19, 25, 34, 98, 41] // smallest element in list[5:] is
41
                        ^
[ 5,  7, 19, 25, 34, 41, 98] // swap it with list[5]
                             // since list[5] is also the
                             // second last element, we are done

[ 5,  7, 19, 25, 34, 41, 98] // END: list is sorted
```

The following code in Python shows one way to do it:

In [ ]:
```
def selectionsort(lis):
    n = len(lis)
    for i in range(n-1):
        smallest = i            # to get the index of the smallest element in
        for j in range(i+1, n):
            if lis[j] < lis[smallest]:   # find the smallest element and get
                smallest = j
        lis[i], lis[smallest] = lis[smallest], lis[i]   # swap the smallest e

mylist = [25, 34, 98, 7, 41, 19, 5]
selectionsort(mylist)
print(mylist)
```

What is the time complexity of this algorithm?

There are two loops, one for `i` and one for `j`. The inner loop runs `n - i` times for each value of `i`, so the total number of times the inner loop runs is

`(n - 1) + (n - 2) + (n - 3) + ... + 2`

times. Since the number of steps in each loop is (roughly) constant, the time complexity of this algorithm is O(n^2).

Bubble Sort

Another natural way to sort a list is to do the following:

1. Compare the zeroth and first elements. If the zeroth is larger than the first, swap them.
2. Compare the first and second elements. If the first is larger than the second, swap them.
3. Compare the second and third elements. If the second is larger than the third, swap them.
4. Going down the list, compare adjacent values as above until you reach the last two elements.

By doing this, the largest element would end up as the last element. This constitutes one pass. We now do a second pass on the list from the zeroth to `n - 1` th element, so that the second largest element moves to the second last position. We then do a third pass to move the third largest element to the third last position, and so on, for a total of `n - 1` passes.

Example

```
[25, 34, 98,  7, 41, 19,  5] // START:
[25, 34, 98,  7, 41, 19,  5] // no swap since 25 < 34
[25, 34, 98,  7, 41, 19,  5] // no swap since 34 < 98
[25, 34,  7, 98, 41, 19,  5] // swap 98 and 7
[25, 34,  7, 41, 98, 19,  5] // swap 98 and 41
[25, 34,  7, 41, 19, 98,  5] // swap 98 and 19
[25, 34,  7, 41, 19,  5, 98] // swap 98 and 5
                             // end of first pass
                             // 98 is in the last position

[25, 34,  7, 41, 19,  5, 98] // no swap since 25 < 34
[25,  7, 34, 41, 19,  5, 98] // swap 7 and 34
[25,  7, 34, 41, 19,  5, 98] // no swap since 34 < 41
[25,  7, 34, 19, 41,  5, 98] // swap 41 and 19
[25,  7, 34, 19,  5, 41, 98] // swap 41 and 5
                             // end of second pass
```

```
                                // 41 is in the second last
position

[ 7, 25, 34, 19,  5, 41, 98] // swap 25 and 7
[ 7, 25, 34, 19,  5, 41, 98] // no swap since 25 < 34
[ 7, 25, 19, 34,  5, 41, 98] // swap 34 and 19
[ 7, 25, 19,  5, 34, 41, 98] // swap 34 and 5
                                // end of third pass
                                // 34 is in the third last position

[ 7, 25, 19,  5, 34, 41, 98] // no swap since 7 < 25
[ 7, 19, 25,  5, 34, 41, 98] // swap 25 and 19
[ 7, 19,  5, 25, 34, 41, 98] // swap 25 and 5
                                // end of fourth pass
                                // 25 is in the fourth last
position

[ 7, 19,  5, 25, 34, 41, 98] // no swap since 7 < 19
[ 7,  5, 19, 25, 34, 41, 98] // swap 19 and 5
                                // end of fifth pass
                                // 19 is in the fifth last position

[ 5,  7, 19, 25, 34, 41, 98] // swap 7 and 5
                                // end of sixth pass
                                // 7 is in the sixth last position

[ 5,  7, 19, 25, 34, 41, 98] // END: list is sorted
```

Try to write your own code to implement bubble sort. A possible pseudocode version is given below.

```
INPUT MyList
MaxIndex ← LENGTH(MyList)

n ← MaxIndex
FOR i ← 1 TO (MaxIndex - 1)
    FOR j ← 1 TO n
        IF MyList[j] > MyList[j+1] // Remember, in Pseudocode,
                                   // the first element of the
                                   // list has index 1, not 0.
           THEN                    // This carries out the swap.
               Temp ← MyList[j]
               MyList[j] ← MyList[j+1]
               MyList[j+1] ← Temp
         ENDIF
     ENDFOR
     n ← n - 1 // The next time you do the inner loop,
               // we do not need to go all the way to the end,
               // since the largest numbers are already in
order.
ENDFOR
```

In [31]:

```python
lis = [25, 34, 98, 7, 41, 19, 5]

def bubblesort(lis):
    n = len(lis) - 1
    for i in range(n):
        for j in range(n):
```

```
            if lis[j] > lis[j+1]:
                lis[j], lis[j+1] = lis[j+1], lis[j]
        n -= 1
    return lis

bubblesort(lis)
```

Out[31]: `[5, 7, 19, 25, 34, 41, 98]`

What is the time complexity of this algorithm?

Similar to selection sort, the number of times the inner loop runs is

$$(n - 1) + (n - 2) + (n - 3) + ... + 2 + 1$$

times. Since the number of steps in each loop is (roughly) constant, the time complexity of this algorithm is also $O(n^2)$.

Insertion Sort

In this particular sort, we consider the list as having a sorted part and an unsorted part. At the beginning, the sorted part only consists of one element. One by one, elements from the unsorted part are inserted into the sorted part in the correct location. At the end of the algorithm, all the elements end up in the sorted part.

Example

In this example, the `^` symbol is the separator between the sorted part and the unsorted part. The sorted part contains all the elements before `^`, and the unsorted part contains the element `^` is pointing to and everything else that comes after it. The element `^` is pointing to is called the **key**. It is the element that we are going to move from the unsorted part to the sorted part.

```
[25, 34, 98,  7, 41, 19,  5] // START:
     ^                       // The sorted part is [25],
                             // and the unsorted part is [34,
98,  7, 41, 19,  5].
                             // We now try to insert 34 into the
sorted part.
                             // Since 25 < 34, we do not need to
do anything.

[25, 34, 98,  7, 41, 19,  5] // The sorted part is [25, 34],
         ^                   // and the unsorted part is [98,
7, 41, 19,  5].
                             // We now try to insert 98 into the
```

```
sorted part.
                              // Since 34 < 98, we do not need to
do anything.

[25, 34, 98,  7, 41, 19,  5] // The sorted part is [25, 34, 98],
             ^                // and the unsorted part is [7, 41,
19,  5].
                              // We now try to insert 7 into the
sorted part.
                              // We do this by swapping 7 with
the element
                              // before it until it is in the
right place,
                              // i.e. the element before 7 is
smaller than it.
                              // In this case, 7 will swap all
the way
                              // to the beginning of the list.

[ 7, 25, 34, 98, 41, 19,  5] // The sorted part is [7, 25, 34,
98],
                  ^           // and the unsorted part is [41,
19,  5].
                              // We now try to insert 41 into the
sorted part.
                              // We do this by swapping 41 with
the element
                              // before it until it is in the
right place,
                              // i.e. the element before 41 is
smaller than it.
                              // In this case, 41 will swap with
98 and stop.

[ 7, 25, 34, 41, 98, 19,  5] // The sorted part is [7, 25, 34,
41, 98],
                     ^        // and the unsorted part is [19,
5].
                              // We now try to insert 19 into the
sorted part.
                              // We do this by swapping 19 with
the element
                              // before it until it is in the
right place,
                              // i.e. the element before 19 is
smaller than it.
                              // In this case, 19 will swap with
98,
                              // then with 41, 32 and 25,
                              // until it ends up between 7 and
25.

[ 7, 19, 25, 34, 41, 98,  5] // The sorted part is [7, 19, 25,
34, 41, 98],
                           ^  // and the unsorted part is [5].
                              // We now try to insert 5 into the
sorted part.
```

```
                                  // We do this by swapping 5 with
    the element
                                  // before it until it is in the
    right place,
                                  // i.e. the element before 5 is
    smaller than it.
                                  // In this case, 5 will swap all
    the way
                                  // to the beginning of the list.

    [ 5,  7, 19, 25, 34, 41, 98] // The unsorted list is now empty.
                                  // END: list is sorted
```

Try to write your own code to implement Insertion sort. A possible pseudocode version is given below.

```
    INPUT MyList
    MaxIndex ← LENGTH(MyList)

    FOR i ← 2 TO MaxIndex // Remember, in Pseudocode,
                          // the first element of the
                          // list has index 1, not 0.
        Key ← MyList[i]
        CompareWithKey ← i-1 // We start comparing with the
                             // element immediately before Key,
    i.e.,
                             // the largest element in the sorted
    part.
        WHILE (MyList[CompareWithKey] > Key) AND (CompareWithKey >
    0) // The swapping

    // happens inside

    // this loop

            MyList[CompareWithKey + 1] ← MyList[CompareWithKey] //
    Swap CompareWithKey
                                                                  //
    with the element
                                                                  //
    after it

            CompareWithKey ← CompareWithKey - 1 //
    MyList[CompareWithKey] moves
                                                 // to the previous
    item

        ENDWHILE // We have reached an element smaller than Key,
                 // or all elements in the sorted part are larger
    than Key

        MyList[CompareWithKey + 1] ← Key // Insert Key
    ENDFOR
```

In [9]:

```
lis = [25, 34, 98, 7, 41, 19, 5]

def insertionsort(lis):
    for i in range(1, len(lis)):
```

```
        key = lis[i]
        compare = i-1
        while lis[compare] > key and compare >= 0:
            lis[compare+1] = lis[compare]
            compare -= 1
        lis[compare+1] = key
    return lis

insertionsort(lis)
```

Out[9]: `[5, 7, 19, 25, 34, 41, 98]`

What is the time complexity of this algorithm?

Similar to the previous two sorting algorithms, we have an inner loop and an outer loop. In the worst case scenario, the inner loop (the WHILE loop) needs to compare Key with all the previous values, which happens when Key is smaller than all the elements in the sorted part. In this case, it will run through `i - 1` cycles of the loop. Therefore, in the worst case scenario, the total number of times the inner loop runs is

```
1 + 2 + 3 + ... + (n - 1)
```

times. Since the number of steps in each loop is (roughly) constant, the time complexity of this algorithm is O(n^2).

Comparing what we have so far...

One common feature of the previous three sorting algorithms (other than the fact that they are O(n^2)) is that they only involve swapping elements within the list. This means that no additional memory is required (except for an occasional `Temp` variable) as we are rearranging data within the memory addresses already allocated to the original list. This also means that the sequence of elements in the original list is destroyed when we carry out the sorting algorithm. This is known as **sorting in place**.

Since all three algorithms are O(n^2) in the worst case scenario, is there any reason for choosing one above the other? We could compare best case scenarios as well. Notice that the loops in bubble sort and selection sort are constant regardless of the arrangement of the elements, so it is still O(n^2). On the other hand, if the original list is already almost sorted, the inner loop of insertion sort would not have to repeat so many times. In the best case scenario, if the list happens to be already sorted, the inner loop would not run at all, and the time complexity drops to O(n).

Does this mean that insertion sort is always better? Yes and no. For a list that is already almost sorted, it would be the fastest algorithm. However, for a random list, the time complexity is the same as the other two. We can look at the number of actual swaps made. The steps in selection sort are mostly comparisons, and it makes at most n-1 swaps. On the other hand, bubble sort and insertion sort could make as many as O(n^2) swaps.

Depending on the nature of the elements in the list, if swapping two elements is extremely time- or memory-consuming (e.g. the elements are very large or the storage device is slow), then selection sort might be the better choice. On the other hand, if you have a reason to believe that the list is already almost sorted - if most elements are somewhat near their final location - then insertion sort might be better.

Is it possible to do better than O(n^2)? Yes, and for that, we would need to use the power of recursion.

Merge Sort

Merge sort is a recursive algorithm that is O(n log n), which is a marked improvement over O(n^2). However, it has two disadvantages. Firstly, the constant coefficient that we hid away in the big-O notation is large, which means that for small values of n, other sorting algorithms might be faster. Secondly, it does not sort in place, i.e. it makes multiple copies of the input list, which takes up more memory. Therefore, merge sort can only be used when there is the luxury of having enough memory for many copies of the list.

Merge sort works on the **divide-and-conquer** approach:

- **Divide**: Split the list into two smaller lists, preferably of almost equal length.
- **Conquer**: Sort each of the two smaller lists. We can do this recursively, by splitting them again into even smaller lists, until we get lists of length 1, which do not need to be sorted.
- **Combine**: Merge the two sorted smaller lists back into a longer sorted list.

Example

```
[25, 34, 98,  7, 41, 19,  5]     // START
       ↓                ↓
[23, 34, 98,  7] [41, 19,  5]    // Divide
   ↓        ↓        ↓      ↓
[23, 34] [98,  7] [41, 19] [5]   // Divide
  ↓   ↓    ↓    ↓   ↓    ↓   ↓
[23] [34] [98] [7] [41] [19] [5] // Divide. All lists are of
length 1.
  ↓   ↓    ↓    ↓   ↓    ↓   ↓
[23, 34] [ 7, 98] [19, 41] [5]   // Combine
   ↓        ↓        ↓      ↓
[ 7, 23, 34, 98] [ 5, 19, 41]    // Combine
       ↓                ↓
[ 5,  7, 19, 23, 34, 41, 98]     // END: list is sorted
```

Try to write your own code to implement Merge sort. A possible pseudocode version is given

below.

```
FUNCTION MergeSort(MyList: LIST) RETURNS LIST
    MaxIndex ← LENGTH (MyList)

    IF MaxIndex > 1
        THEN
            Half ← MaxIndex DIV 2
            LeftList ← LEFT(MyList, Half)
            RightList ← RIGHT(MyList, Half)

            SortedLeftList ← MergeSort(LeftList)
            SortedRightList ← MergeSort(RightList)

            Result ← Merge(SortedLeftList, SortedRightList)

        ELSE
            Result ← MYLIST
    ENDIF

    RETURN Result
ENDFUNCTION
```

Note that we have made use of **wishful thinking**, using another function `Merge` that merges two sorted lists into one.

```
FUNCTION Merge(MyList1 : LIST[0:M-1], MyList2 : LIST[0:N-1])
RETURNS LIST
    Length1 ← LENGTH(MyList1)
    Length2 ← LENGTH(MyList2)
    TotalLength ← Length1 + Length2

    DECLARE Result : LIST[0:M+N-1] // We are going to RETURN
Result.
                                   // It is a list of length M+N
    Pos1 ← 0
    Pos2 ← 0
    PosResult ← 0

    WHILE Pos1 < M AND Pos2 < N
        IF MyList1[Pos1] <= MyList2[Pos2]
```

```
                THEN
                    Result[PosResult] ← MyList1[Pos1]
                    Pos1 ← Pos1 + 1
                    PosResult ← PosResult + 1
                ELSE
                    Result[PosResult] ← MyList1[Pos2]
                    Pos2 ← Pos2 + 1
                    PosResult ← PosResult + 1
            ENDIF
        ENDWHILE

        IF Pos1 = M
            THEN
                FOR X ← Pos2 TO N-1
                    Result[PosResult] ← MyList2[X]
                    PosResult ← PosResult + 1
                ENDFOR
            ELSE
                FOR X ← Pos1 TO M-1
                    Result[PosResult] ← MyList1[X]
                    PosResult ← PosResult + 1
                ENDFOR
        ENDIF

        RETURN Result
    ENDFUNCTION
```

Try to write your own code to implement Merge sort.

In [16]:
```python
lis = [25, 34, 98,  7, 41, 19,  5]

def merge(lis1, lis2):   # Assume lis1 and lis2 are already sorted
    result = []

    while len(lis1) > 0 and len(lis2) > 0:
        if lis1[0] <= lis2[0]:
            result.append(lis1.pop(0))
        else:
            result.append(lis2.pop(0))

    if not lis1:
        result.extend(lis2)
    else:
        result.extend(lis1)

    return result

def mergesort(lis):
    if len(lis) > 1:
        half = len(lis) // 2
        left, right = lis[:half], lis[half:]
        sleft, sright = mergesort(left), mergesort(right)
        result = merge(sleft, sright)
    else:
        result = lis
    return result

mergesort(lis)
```

Out[16]:
```
[5, 7, 19, 25, 34, 41, 98]
```

It can be shown that merge sort is O(n log n).

However, as we can see, it creates many new lists in the course of carrying out the function. Therefore, it draws heavily on the computer's memory.

The merge sort algorithm was created by John von Neumann in 1948. In the 1980s and 1990s, other researches developed algorithms that perform merge sort in place (and hence do not use an exponential amount of memory). These algorithms are beyond the scope of the A-level syllabus.

Quicksort

Quicksort is another recursive sorting algorithm that uses a divide-and-conquer approach. Unlike the merge sort algorithm described above, quicksort can be performed in place. The main idea behind quicksort is as follows:

- **Divide**: Choose one element called the **pivot**. Rearrange the list so that all elements smaller than the pivot are on the left of it, and all elements larger than the pivot are on the right of it.
- **Conquer**: Sort each of the two smaller lists. We can do this recursively, by splitting them again into even smaller lists until we get lists of length 1, which do not need to be sorted.
- **Combine**: Since the sort was performed in place, we do not need to do anything further!

A rough demonstration of the idea is shown using the Python code below. Note that this is not a full implementation since it does not work in place. (It creates two new lists in each step of the recursion.) It does, however, work recursively.

In [ ]:
```
lis = [20, 47, 12, 53, 31, 84, 85, 96, 45, 18]

def quicksort(lis):
    n = len(lis)
    if n == 0 or n == 1:
        return lis
    else:
        lolist = []
        hilist = []
        pivot = lis[0]
        for i in range(1, n):
            if lis[i] < pivot:
                lolist.append(lis[i])
            else:
                hilist.append(lis[i])
        return quicksort(lolist) + [pivot] + quicksort(hilist)

print(quicksort(lis))
```

We shall see two possible implementations of quicksort.

The **first** is not the original implementation, but may be easier to understand and implement.

The main difficulty in the algorithm is the 'Divide' part. We will create a procedure called `Partition` to do this. We divide the list into 4 sections:

1. Elements smaller than the pivot (starting from index `0`);
2. Elements larger than the pivot (starting from index `i`);
3. Elements which we do not know about yet (starting from index `j`);

4. The last element, which is the pivot.

At the beginning, the first two sections are empty, and all elements except the last (the pivot) are in the third section. Therefore, `i` and `j` are both 0. We then look at the `j` th element in the list (the start of the third section) and compare it with the pivot.

- If it is larger than the pivot, we simply increment `j` by 1, moving that element from the third section to the second, without actually changing its location.
- If it is smaller than the pivot, we swap it with element with index `i`, and then increment both `i` and `j` by 1. This moves the element from the third section to the end of the first section. We carry this out until the third section is empty, that is, `j` is equal to `n–1`.

Now, our list consists of elements smaller than the pivot, followed by elements larger than the pivot, followed by the pivot itself. We swap the pivot with the element with index `i` (the start of the second section), ensuring that our pivot is at the correct location, between the two sections.

The algorithm is written below in pseudocode. Notice that the indices of `MyList` run from `L` to `R` (instead of the more conventional 0 to `n–1`, or even 1 to `n`). This is because we will eventually have to carry out this procedure on a section of the original list, so the indices themselves are also variables.

```
FUNCTION Partition(L : INTEGER, R : INTEGER, MyList : LIST)
RETURNS INTEGER
    Pivot ← MyList[R]
    i ← L
    j ← L

    REPEAT
        IF MyList[j] > Pivot
            THEN
                j ← j + 1
            ELSE // swap elements with index i and j
                Temp ← MyList[j]
                MyList[j] ← MyList[i]
                MyList[i] ← Temp
                i ← i + 1
                j ← j + 1
        ENDIF
    UNTIL j = R

    MyList[R] ← MyList[i] // swap elements with index i and R
(the pivot)
    MyList[i] ← Pivot

    RETURN i
ENDFUNCTION
```

While the function does the work of sorting out the list, it also returns the index of the pivot as an integer. This will be useful when we use the function to perform the quicksort later.

We can now carry out the 'Conquer' and 'Combine' parts of the algorithm.

```
PROCEDURE Quicksort(L : INTEGER, R : INTEGER, MyList : LIST)
    IF R-L >= 1 // if the list has 0 or 1 element (R - L < 1),
                // then it is automatically already sorted.
        THEN
            PivotPos = Partition(L, R, MyList)
            CALL Quicksort(L, PivotPos - 1, MyList)
            CALL Quicksort(PivotPos + 1, R, MyList)
    ENDIF
ENDPROCEDURE
```

Try to write your own code to implement quicksort.

In [8]:
```python
lis = [20, 47, 12, 53, 31, 84, 85, 96, 45, 18]

def partition(lis, L, R):
    pivot = lis[R]
    left, right = L, L

    while right != R:
        if lis[right] <= pivot:
            lis[left], lis[right] = lis[right], lis[left]
            left += 1
        right += 1
    lis[R], lis[left] = lis[left], pivot
    return left

def quicksort(lis, L = None, R = None):
    if None in (L, R):
        L, R = 0, len(lis) - 1
    if R - L >= 1:
        pivot = partition(lis, L, R)
        quicksort(lis, L, pivot - 1)
        quicksort(lis, pivot + 1, R)
    return lis

lis = [20, 47, 12, 53, 31, 84, 85, 96, 45, 18]
print(quicksort(lis))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The **second** implementation is the original one described by Sir Charles Antony Richard Hoare when he published the Quicksort algorithm in 1959. It differs from the first in the implementation of Partition. It divides the list into only 3 sections:

1. 'Small' elements;
2. Elements which we do not know about yet;
3. 'Large' elements.

It makes use of two indices which start at the two ends of the list and gradually move towards each other until they meet in the middle. These two indices separate the three sections of the list. When they finally meet, the second section is empty, and the element to which both indices point is the pivot.

At the beginning, the first and third sections are empty, so `i` is 0 and `j` is `n` -1.

- Look at the element with index `i` and the element with index `j` . If the smaller element is on the right, we swap them, and then we swap the values of `i` and `j` , and then move `j` closer to `i` by 1. (If `j > i` , we decrease `j` by 1. If `j < i` , we increase `j` by 1.)

- If the smaller element is on the left, we move `j` closer to `i` by 1.

Eventually `i` and `j` will coincide, and the element with that index becomes the pivot. The algorithm is written below in pseudocode.

```
FUNCTION Partition(L : INTEGER, R : INTEGER, MyList : LIST)
RETURNS INTEGER
    i ← L
    j ← R
    REPEAT
        IF j > i
            THEN
                IF MyList[j] < MyList[i]
                    THEN
                        // swap MyList[j] and MyList[i]
                        Temp ← MyList[j]
                        MyList[j] ←  MyList[i]
                        MyList[i] ← Temp

                        // swap j and i
                        Temp ← j
                        j ← i
                        i ← Temp

                        // now j < i, so to move j closer to i,
add 1
                        j ← j + 1
                    ELSE
                        j ← j - 1
                ENDIF
            ELSE // when j < i
                IF MyList[j] > MyList[i]
                    THEN
                        // swap MyList[j] and MyList[i]
                        Temp ← MyList[j]
                        MyList[j] ←  MyList[i]
                        MyList[i] ← Temp

                        // swap j and i
                        Temp ← j
                        j ← i
                        i ← Temp

                        // now j > i, so to move j closer to i,
subtract 1
                        j ← j - 1
                    ELSE
                        j ← j + 1
                ENDIF
        ENDIF
    UNTIL j = i
    RETURN i
ENDFUNCTION
```

The 'Conquer' and 'Combine' parts of the algorithm are the same as in the previous implementation.

Try to write your own code to implement this version of Quicksort.

In [30]:
```
lis = [20, 47, 12, 53, 31, 84, 85, 96, 45, 18]

def partition(lis, L, R):
    i, j = L, R
    while j != i:
        if j > i:
            if lis[j] < lis[i]:
                lis[i], lis[j] = lis[j], lis[i]
                i, j = j, i + 1
            else:
                j -= 1
        else:
            if lis[j] > lis[i]:
                lis[i], lis[j] = lis[j], lis[i]
                i, j = j, i - 1
            else:
                j += 1
    return i

def quicksort(lis, L = None, R = None):
    if None in (L, R):
        L, R = 0, len(lis) - 1
    if R - L >= 1:
        ppos = partition(lis, L, R)
        quicksort(lis, L, ppos - 1)
        quicksort(lis, ppos + 1, R)
    return lis

quicksort([3, 4, 5, 5, 6, 7, 8, 9])
```

Out[30]:
```
[3, 4, 5, 5, 6, 7, 8, 9]
```

It can be shown that both implementations of Quicksort are $O(n^2)$ in the worse case scenario. However, if we consider all possible permutations of a list and take the average, it turns out that on average, it is $O(n \log n)$. In other words, a randomly chosen arrangement of the list can be be sorted in $O(n \log n)$ time, on average. This, combined with the fact that quicksort works in place, makes it one of the most popular sorting algorithms.

Summary

These are only a handful of the numerous sorting algorithms that have been developed over the decades. There is no one best method for sorting because they differ so much in terms of time complexity, memory use and ease of implementation. A summary of the time complexities of the various sorting algorithms is given below.

Algorithm	In place?	Time complexity (worst case)	Time complexity (best case) (not in syllabus)	Time complexity (average case) (not in syllabus)
Selection sort	Y	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	Y	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	Y	$O(n^2)$	$O(n)$	$O(n^2)$
Merge sort	N	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	Y	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

So what does Python do?

The `sort` function and the `sorted` method in Python are implemented using Timsort, named after Tim Peters, one of the main developers of Python. It is a hybrid between insertion sort and merge sort. Firstly, it looks for **runs**, sequences of increasing or decreasing numbers that already occur inside the list naturally. Since Python was developed to deal with real-world data, such sequences can be expected to occur with fairly high frequency. While most runs may be short, there could be a few long ones. Python uses Insertion sort to combine the shorter runs before merging them together. Timsort is O(n log n) in the worse case and O(n) in the best case scenario.

Interactive applets

- https://csfieldguide.org.nz/en/interactives/sorting-algorithms/
- https://visualgo.net/en/sorting
- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Further reading

- https://classic.csunplugged.org/sorting-algorithms/
- https://www.toptal.com/developers/sorting-algorithms
- http://computationaltales.blogspot.com/2011/04/why-tailors-use-insertion-sort.html

Youtube videos

- https://www.youtube.com/watch?v=Ns4TPTC8whw (Selection sort)
- https://www.youtube.com/watch?v=yIQuKSwPIro (Bubble sort)
- https://www.youtube.com/watch?v=lyZQPjUT5B4 (Bubble sort)
- https://www.youtube.com/watch?v=6Gv8vg0kcHc (Bubble sort)
- https://www.youtube.com/watch?v=ROalU379l3U (Insertion sort)
- https://www.youtube.com/watch?v=pcJHkWwjNI4 (Insertion sort and Selection sort compared)
- https://www.youtube.com/watch?v=XaqR3G_NVoo (Merge sort)
- https://www.youtube.com/watch?v=KF2j-9iSf4Q (Merge sort)
- https://www.youtube.com/watch?v=MZaf_9IZCrc (Quicksort)
- https://www.youtube.com/watch?v=ywWBy6J5gz8 (Quicksort)
- https://www.youtube.com/watch?v=XE4VP_8Y0BU (Quicksort)
- https://www.youtube.com/watch?v=SLauY6PpjW4 (Quicksort)
- https://www.youtube.com/watch?v=kgBjXUE_Nwc (Sorting and Big-O notation)

Some of these videos may discuss how to implement the algorithms in languages other than Python. The basic underlying ideas are the same, but they are expressed using different syntax.

2020 JC1 H2 Computing 9569

23. Searching Algorithms

Besides sorting that has been covered in the previous chapter, another basic task a computer scientist needs to be able to do is **search**, that is, to retrieve information stored within some data structures.

Linear Search

Suppose we take an integer as an input, and we wish to search for this integer in an existing array (list).

Starting from the first element in the array, we check each element in turn until the search value is found or we reach the end of the array. This is called a **linear search**.

Identifier	Explanation
`MyList`	Data structure (array) to store seven integers
`MaxIndex`	The number of elements in the array
`SearchValue`	The value we are searching for
`Found`	`TRUE` if the value is found `FALSE` if the value is not found
`Index`	Index of the array element currently being processed

A pseudocode algorithm for linear search is given below. (In this case, the array index starts from 1.)

```
INPUT SearchValue
MaxIndex ← LENGTH(MyList)
Found ← FALSE
Index ← 0

REPEAT
    Index ← Index + 1
    IF MyList[Index] = SearchValue
        THEN
            Found ← TRUE
    ENDIF
UNTIL FOUND = TRUE OR Index >= MaxIndex

IF Found = TRUE
    THEN
        OUTPUT "Value found at location:" Index
    ELSE
        OUTPUT "Value not found"
ENDIF
```

The condition in the `REPEAT...UNTIL` loop allows us to exit the loop once the search element is found. Using `Found` makes it easier to read; it is initialised to `FALSE` before

entering the loop and set to `TRUE` once the value is found.

If the value is not in the array, the loop terminates when `Index` is greater than or equal to `MaxIndex`, which happens when we reach the end of the array.

Try writing the algorithm in Python.

In [37]:
```
import time

MyList = [1, 6, 3, 2, 7, 8, 4, 9, 5]

def linear_search(item, lst):
    index = 0
    while index < len(lst):
        if lst[index] == item:
            return True, index
        index += 1
    return False

def linear_search_2(item, lst):
    index = 0
    lst.append(item)
    while True:
        if lst[index] == item:
            break
        index += 1
    del lst[index]
    if index == len(lst):
        return False
    else:
        return True, index

start = time.time()
print(linear_search(7, MyList))
print(linear_search(5, MyList))
print(linear_search(0, MyList))
print(time.time()-start)

start = time.time()
print(linear_search_2(7, MyList))
print(linear_search_2(5, MyList))
print(linear_search_2(0, MyList))
print(time.time()-start)
```

```
(True, 4)
(True, 8)
False
0.0004279613494873047
(True, 4)
(True, 7)
False
0.0002028942108154297
```

Binary Search

If we know that the list we are searching from has been sorted, we can use a different technique.

We can look at the middle entry in the list. For example, if the search value is larger than the middle entry, we only need to search in the second half of the list. After that, we look at the middle entry in the second half (i.e. approximately at the 3/4 mark of the original list). If the search value is smaller than this, we know that it is in the third quarter of the original list. By

continuing to halve the section of the list, we eventually narrow it down until we find the search value or discover that it is not in the list. Since we keep dividing the list into two sections, this is known as a **binary search**.

A pseudocode algorithm for binary search is given below. (In this case, the array index starts from 1.)

```
INPUT SearchValue
Found ← FALSE
SearchFailed ← FALSE
Left ← 1
Right ← MaxIndex

WHILE NOT Found AND NOT SearchFailed
    Middle ← (Left + Right) DIV 2
    IF MyList[Middle] = SearchValue
        THEN
            Found ← TRUE
        ELSE
            IF Left >= Right
                THEN
                    SearchFailed ← TRUE
                ELSE
                    IF MyList[Middle] > SearchValue
                        THEN
                            Right ← Middle - 1
                        ELSE
                            Left ← Middle + 1
                    ENDIF
            ENDIF
    ENDIF
ENDWHILE

IF Found = TRUE
    THEN
        OUTPUT "Value found at location:" Middle
    ELSE
        OUTPUT "Value not found"
ENDIF
```

Try writing the algorithm in Python.

In [6]:
```python
MyList = [1, 2, 3, 4, 5, 6, 7, 8, 9]

def binary_search(item, lst):
    found, failed = False, False
    left, right, middle = 0, len(lst), None
    while (not found) and (not failed):
        middle = (left + right) // 2
        if lst[middle] == item:
            found = True
        elif left >= right:
            failed = True
        elif lst[middle] > item:
            right = middle - 1
        else:
            left = middle + 1
    if found == True:
        return True, middle
```

```
    elif failed == True:
        return False

print(binary_search(7, MyList))
print(binary_search(5, MyList))
print(binary_search(0, MyList))
```

```
(True, 6)
(True, 4)
False
```

What is the time complexity of linear search and binary search in the worst case scenario?

Hash Table Search

Hash Table

Another method of storing data in an array so that we can access them easily is to use the concept of a **hash table**.

Comprising a **key** and the **record**, every piece of data in a hash table has a unique key. For example, student data can use the NRIC number as the key, and the record can contain the name, class, etc.

The idea behind the hash table is to store the value in such a way that we can calculate the index, or **address**, from the key. When we wish to search for a record, we calculate the address from the key and go to that address to access the record. The process of calculating the index from the key is called **hashing**.

Ideally, we would choose a function such that every key value would give a different address (mathematically known as a one-one function). However, this may not always be possible. If two different key values inadvertently hash to the same address, this creates a **collision**. There are some ways to handle collisions:

- Chaining: Create a linked list for collisions with the start pointer at the hashed address.
- Using overflow areas: All collisions are stored in a separate area reserved for overflows. This is known as **closed hashing**.
- Using neighbouring slots: Perform a linear search from the hashed address to find a nearby empty slot. This is known as **open hashing**.

Example

A shop wishes to store customer records in an array. For simplicity, suppose that the array has 10 slots with indexes from 0 to 9. Each customer has a unique customer ID, which is a five-digit number from 10001 to 99999.

One possible hashing function (not a very good one!) is to find the remainder when the ID is divided by 10. This gives us a number from 0 to 9, which can be the index.

```
FUNCTION Hash(Key) RETURNS INTEGER
    Address ← Key MOD 10
    RETURN Address
ENDFUNCTION
```

Suppose we have three customers with IDs 45876, 32390 and 95312. They can be stored in indexes 6, 0 and 2 respectively.

Subsequently, when the shop is going to store a customer record with ID 64636, there will be a collision at index 6. If this customer's record is to be stored at index 6, it will overwrite the record for the customer with ID 45876. To resolve the collision using **open hashing**, the customer record with ID 64636 can be stored in the next available space, which is at index 7.

If another customer record with ID 23467 is to be stored, since the space at index 7 has been taken up by ID 64636, it can be stored in the next available space, which is at index 8.

When searching for records, we need to take into account the fact that some key values may not be at the correct addresses. Adjacent addresses have to be searched too when open hashing is applied. On the other hand, we also know that the record we are searching for does not exist if the key hashes to a location that is empty.

The following pseudocode shows how to insert a record into a hash table, and search for a record using the key. The hash table has indexes from `0` to `Max`.

```
PROCEDURE Insert(NewRecord)
    Index ← Hash(NewRecord.Key)
    WHILE HashTable[Index] IS NOT empty  // in the event of
collision(s)
        Index ← Index + 1                // go to next slot in
the array
        IF Index > Max                   // at the end of the
array,
            THEN                         // wrap around to the
beginning
                Index ← 0
        ENDIF
    ENDWHILE
    HashTable[Index] ← NewRecord         // insert the record
ENDPROCEDURE

FUNCTION FindRecord(SearchKey) RETURNS Record
    Index ← Hash(SearchKey)
    WHILE (HashTable[Index].Key <> SearchKey) AND
(HashTable[Index] IS NOT empty)
        Index ← Index + 1                // go to next slot
        IF Index > Max                   // at the end of the
array,
            THEN                         // wrap around to the
beginning
                Index ← 0
        ENDIF
    ENDWHILE
    IF HashTable[Index] IS NOT empty     // if the record found
        THEN
            RETURN HashTable[Index]      // return the record
    ENDIF
ENDFUNCTION
```

Dictionary

Python **dictionary** is a data type that is an unordered collection of key-value pairs. The key is the term used to look up the required value. (For example, in a real-life dictionary, the key is the word and the value is the definition.) It is defined using curly braces `{ }`. The key-value pairs are separated using a colon `:`. The values can be any data types, but the keys must be immutable, such as integers, strings or tuples.

A dictionary is implemented using a hash table, so that values can be accessed directly by hashing the key.

As an example, an English-Malay dictionary is shown, where the key is the word in English and the value is the corresponding Malay word.

In [ ]:
```
dic = {'sun':'hari', 'moon':'bulan', 'fire':'api', 'water':'air', 'wind':'ang
print(dic)
```

In [ ]:
```
print(dic['moon'])       # look up a value using the key
```

In [ ]:
```
dic['sky'] = 'langit'    # add a new key-value pair
print(dic)
```

In [ ]:
```
dic['long'] = 'panjang'  # change or update a value
print(dic)
```

Note that the key types may be different, as long as they are immutable.

In [ ]:
```
demo = {2:['a','b','c'], (2,4): 27, 'x':{1:2.5, 'a':3}} # the value can be an
print(demo)
```

In [ ]:
```
print(demo[2])           # look up values using the keys
print(demo[(2,4)])
print(demo['x'])
```

In [ ]:
```
print(demo['x'][1])      # look up a value inside the inner dictionary using i
```

In [ ]:
```
print('a' in demo)       # do these keys exist?
print('x' in demo)
```

In [ ]:
```
print(len(demo))         # get the number of key-value pairs
```

In [ ]:
```
for key in demo:         # iterates through the keys
    print(key)           # and display them

for key in demo:         # iterates through the keys
    print(demo[key])     # and display the values
```

In [ ]:
```
items = demo.items())    # get all the key-value pairs
keys = demo.keys())      # get all the keys as a list
values = demo.values()) # get all the values as a list

print(tuple(items))      # display all the key-value pairs as a tuple
print(list(keys))        # display all the keys as a list
print(list(values))      # display all the values as a list
```

2021 JC2 H2 Computing 9569
24. Relational Database: SQL

Introduction

Imagine a situation where you are part of the school administration team in the olden days, having to manage hundreds and thousands of physical files of staff and student records in multiple cabinets.

What are some of the issues that you may encounter?

With the advancement of technology, we no longer have to keep and manage physical records.

A **database** is a collection of data stored in an organised or logical manner. Storing data in a database allows us to access and manage the data. Some examples of databases in real-life include student records, supermarket inventory and contact list.

In general, there are two types of databases: **relational** and **non-relational**. In this chapter, we shall look into the former.

A **relational (SQL) database** is a collection of relational tables with a fixed **schema**, which is the precise description of the data to be stored and the relationships between them. In this model, the data are stored in relational tables and represented in the form of tuples as follows.

```
<TableName>(<Field1>, <Field2>, …)
```

Attributes of Relational Database

A **table** is a two-dimensional representation of data stored in rows and columns. Each table is made up of records and fields.

Below is an example of a table called `StudentMD10`, showing data of students from an imaginary form class MD10.

RegNo	Name	Gender	MobileNo
1	Adam	M	92313291
2	Adrian	M	92585955
3	Agnes	F	83324112
4	Aisha	F	88851896
5	Ajay	M	94191061
6	Alex	M	98675171
7	Alice	F	95029176
8	Amy	F	98640883
9	Andrew	M	95172444
10	Andy	M	95888639

A **record** is a complete set of data about a single entity in the table. In the table above, there are 10 records, each referring to the complete set of data of a particular student.

A **field** or **column** refers to one type of data about the entities in the table. In the table above, there are 4 fields: `RegNo`, `Name`, `Gender` and `MobileNo`.

Quick Check

Express the table `StudentMD10` using the tuple representation mentioned in the previous page.

```
StudentMD10(RegNo, Name, Gender, MobileNo)
```

Keys in Relational Database

A **candidate key** is a minimal set of fields that can uniquely identify each record in a table. It should never be empty.

A **primary key** is a candidate key that is most appropriate to become the main key for a table. It uniquely identifies each record in a table and should not change over time. That is, a primary key tells a particular record apart from another record.

Quick Check
Which of the fields in the table `StudentMD10` is a suitable primary key?

`RegNo`

A **secondary key** is a candidate key that is not selected as a primary key.

A **composite key** is a combination of two or more fields in a table that can be used to uniquely identify each record in a table. Uniqueness is only guaranteed when the fields are combined. When taken individually, the fields do not guarantee uniqueness.

Quick Check
A table called `StudentMD1011` is shown below.

RegNo	Name	Gender	FormClass
1	Adam	M	MD10
2	Adrian	M	MD10
3	Agnes	F	MD10
4	Aisha	F	MD10
5	Ajay	M	MD10
6	Alex	M	MD10
7	Alice	F	MD10
8	Amy	F	MD10
9	Andrew	M	MD10
10	Andy	M	MD10
1	Adam	M	MD11
2	Bala	M	MD11
3	Bee Lay	F	MD11
4	Ben	M	MD11
5	Boon Kiat	M	MD11
6	Boon Lim	M	MD11
7	Chee Seng	M	MD11
8	Colin	M	MD11
9	Daniel	M	MD11
10	Eleanor	F	MD11

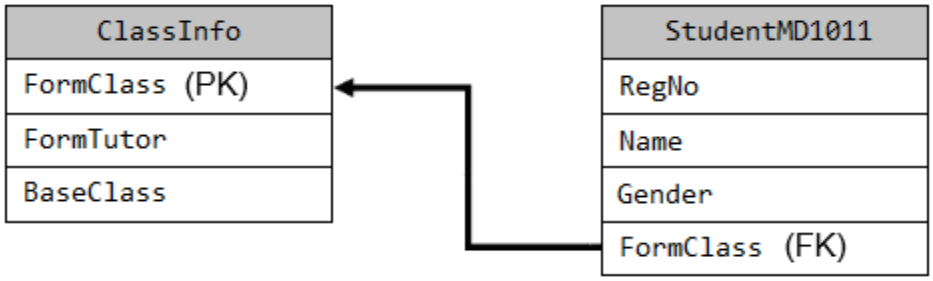
Which two fields form the composite key for the table? `RegNo` and `FormClass`

A **foreign key** is a field in one table that refers to the primary key in another table.

To illustrate this concept, take a look at another table below called `ClassInfo` with `FormClass` chosen to be the primary key.

FormClass	FormTutor	BaseClass
MD10	Peter Lim	F3.1
MD11	Susan Tan	F3.2

Notice that the primary key (PK) in the table `ClassInfo` is related or linked to the `FormClass` field in table `StudentMD1011`. This makes `FormClass` in the table `StudentMD1011` a foreign key (FK).



Data Redundancy

Data redundancy refers to the same data being stored more than once.

Take a look at the table below.

RegNo	Name	Gender	FormClass	FormTutor
1	Adam	M	MD10	Peter Lim
2	Adrian	M	MD10	Peter Lim
3	Agnes	F	MD10	Peter Lim
4	Aisha	F	MD10	Peter Lim
5	Ajay	M	MD10	Peter Lim
6	Alex	M	MD10	Peter Lim
7	Alice	F	MD10	Peter Lim
8	Amy	F	MD10	Peter Lim
9	Andrew	M	MD10	Peter Lim
10	Andy	M	MD10	Peter Lim

As we can see, the data for `FormClass` and `FormTutor` are repeated for students who are in the same form class. This may lead to potential issues on insertion, updating and deletion of data, such as:

Insertion	A new student cannot be inserted unless a form class and a form tutor have been assigned.
Update	Should Mr Peter Lim quit the school, all the records in the table would need to be updated. Should we miss any record, it would lead to inconsistent data.
Deletion	Should all the records in the table be deleted, information on form class and form tutor would be lost.

Data Dependency

Suppose we have the following table:

```
Student(MatricNo, Name, Gender, FormClass, FormTutor)
```

`MatricNo` is a unique number assigned to every student in the college.

Functional dependency

Attribute Y is **functionally dependent** on attribute X (usually the primary key), if for every valid instance of X, the value of X uniquely determines the value of Y, i.e. X → Y.

`MatricNo` uniquely identifies `Name` because if we know the `MatricNo`, we can know the `Name` associated with it. Therefore, we can say `Name` is functionally dependent on `MatricNo`, i.e.

`MatricNo` → `Name`

Transitive dependency

A functional dependency is said to be **transitive** if it is indirectly formed by two functional dependencies. Z is transitively dependent on X if Y is functionally dependent on X, but X is not functionally dependent on Y, and Z is functionally dependent on Y. In other words, X → Z is a transitive dependency if the following hold true:

- X → Y
- Y does not → X
- Y → Z

`FormClass` is functionally dependent on `MatricNo`, but `MatricNo` is not functionally dependent on `FormClass`, i.e.

`MatricNo` → `FormClass`

On the other hand, `FormTutor` is functionally dependent on `FormClass`, i.e.

`FormClass` → `FormTutor`

Therefore, we can conclude that `FormTutor` is transitively dependent on `MatricNo`, i.e.

`MatricNo` → `FormTutor`

Normalisation

Normalisation is the process of organising the tables in a database to reduce data redundancy and prevent inconsistent data. There are at least three normal forms:

- first normal form (1NF)
- second normal form (2NF)
- third normal form (3NF)

First Normal Form (1NF)

For a table to be in 1NF, all columns must be **atomic**, i.e. the information cannot be broken down further.

Consider the following table.

MatricNo	Name	Gender	Form Class	Form Tutor	Base Class	CCAInfo
1	Adam	M	MD10	Peter Lim	F3.1	Tennis Teacher IC = Adrian Tan
2	Adrian	M	MD10	Peter Lim	F3.1	Choir Teacher IC = Sanjay Vittal, Art Club Teacher IC = Nur Fauziah
3	Adam	M	MD11	Susan Tan	F3.2	Rugby Teacher IC = Zoe Lim
4	Bala	M	MD11	Susan Tan	F3.2	Tech Council Teacher IC = Lilian Phua
5	Bee Lay	F	MD11	Susan Tan	F3.2	Choir Teacher IC = Sanjay Vittal, Chess Teacher IC = Edison Poh

For this example, assume that every form class has only one form tutor, and each CCA has only one teacher IC.

The table above is not in 1NF because the `CCAInfo` column contains multiple values.

In order for the table to be in 1NF, we can split `CCAInfo` into two single-value columns: `CCAName` and `CCATeacherIC`. Notice that the students with `MatricNo` 2 and 5 have multiple CCAs. We keep this information intact by splitting their records into multiple records, each corresponding to a different CCA. The resulting table is shown below.

Matric No	Name	Gender	Form Class	Form Tutor	Base Class	CCA Name	CCA TeacherIC
1	Adam	M	MD10	Peter Lim	F3.1	Tennis	Adrian Tan
2	Adrian	M	MD10	Peter Lim	F3.1	Choir	Sanjay Vittal
2	Adrian	M	MD10	Peter Lim	F3.1	Art Club	Nur Fauziah
3	Adam	M	MD11	Susan Tan	F3.2	Rugby	Zoe Lim
4	Bala	M	MD11	Susan Tan	F3.2	Tech Council	Lilian Phua
5	Bee Lay	F	MD11	Susan Tan	F3.2	Choir	Sanjay Vittal
5	Bee Lay	F	MD11	Susan Tan	F3.2	Chess	Edison Poh

The values for `CCAName` and `CCATeacherIC` are now atomic for each record.

The primary key for the above table shall be the composite key formed by `MatricNo` and `CCAName`.

Second Normal Form (2NF)

For a table to be in 2NF, it must satisfy two conditions:

- The table should already be in 1NF.
- Every non-key attribute must be fully dependent on the entire primary key. This means no attribute can depend on part of the primary key only.

`Name`, `Gender`, `FormClass`, `FormTutor` and `BaseClass` is dependent on only part of the primary key, `MatricNo`.

`CCATeacherIC`, on the other hand, is dependent only on `CCAName`.

Thus, we decompose the 1NF table into three tables shown below.

`Student`

MatricNo	Name	Gender	FormClass	FormTutor	BaseClass
1	Adam	M	MD10	Peter Lim	F3.1
2	Adrian	M	MD10	Peter Lim	F3.1
2	Adrian	M	MD10	Peter Lim	F3.1
3	Adam	M	MD11	Susan Tan	F3.2
4	Bala	M	MD11	Susan Tan	F3.2
5	Bee Lay	F	MD11	Susan Tan	F3.2
5	Bee Lay	F	MD11	Susan Tan	F3.2

`StudentCCA`

MatricNo	CCA Name
1	Tennis
2	Choir
2	Art Club
3	Rugby
4	Tech Council
5	Choir
5	Chess

`CCAInfo`

CCA Name	CCA TeacherIC
Tennis	Adrian Tan
Choir	Sanjay Vittal
Art Club	Nur Fauziah
Rugby	Zoe Lim
Tech Council	Lilian Phua
Choir	Sanjay Vittal
Chess	Edison Poh

Quick Check

What should be the primary or composite key for each of the three tables above?

The primary key for table `Student` should be `MatricNo`.

The composite key for table `StudentCCA` should be `MatricNo` and `CCAName`.

The primary key for table `CCAInfo` should be `CCAName`.

Third Normal Form (3NF)

For a table to be in 3NF, it must satisfy two conditions:

- The table should already be in 2NF.
- The table should not have transitive dependencies.

Quick Check
Explain the transitive dependency found in the `Student` table.

`FormTutor` and `BaseClass` are dependent on `FormClass` and `FormClass` is dependent on `MatricNo`. Therefore, `FormTutor` and `BaseClass` are transitively dependent on `MatricNo`.

To remove the transitive dependency, we decompose the 2NF `Student` table into two tables shown below.

`Student`

MatricNo	Name	Gender	Form Class
1	Adam	M	MD10
2	Adrian	M	MD10
3	Adam	M	MD11
4	Bala	M	MD11
5	Bee Lay	F	MD11

`FormInfo`

Form Class	Form Tutor	Base Class
MD10	Peter Lim	F3.1
MD11	Susan Tan	F3.2

`MatricNo` remains the primary key for the `Student` table.
`FormClass` shall be the primary key of the newly formed table called `FormInfo`.

The final design after normalisation is represented below.

`Student(MatricNo, Name, Gender, FormClass)`

`FormInfo(FormClass, FormTutor, BaseClass)`

`StudentCCA(MatricNo, CCAName)`

`CCAInfo(CCAName, CCATeacherIC)`

The primary key for each table is indicated by underlining one or more attributes.

Each foreign key is indicated by using a dashed underline.

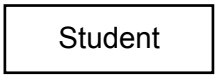
Note:
In the H2 Computing 9569 syllabus, candidates are required to reduce data redundancy to 3NF only. Nevertheless, going through 1NF and 2NF may help in some situations.

Entity-Relationship (E-R) Diagram

An **entity-relationship (E-R) diagram** is a data modelling technique that illustrates the entities of a database and the relationships among those entities. It is useful in the planning of the design of relational databases.

For the purpose of the syllabus, we shall only cover a simplified convention for the drawing of E-R diagrams using **crow's foot notation**.

An **entity** is a specific object of interest. Nouns are usually used to name entities. Entities are represented by rectangles.

e.g. 

A **relationship** describes the link between two entities. One of the following relationships can exist between two entities:

- **one-to-one**

  For example, at a concert with reserved seating, each ticket entitles someone to a particular seat and each seat is linked to only one ticket.

- **one-to-many**

  For example, a form class can have many students, but a student can belong to only one form class.

- **many-to-many**

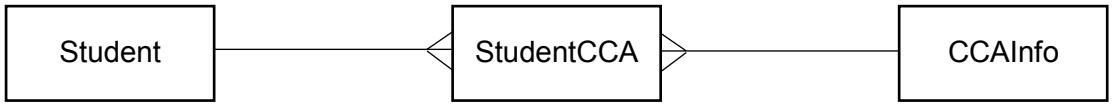
  

  For example, a CCA can have many students, and a student can join many CCAs.

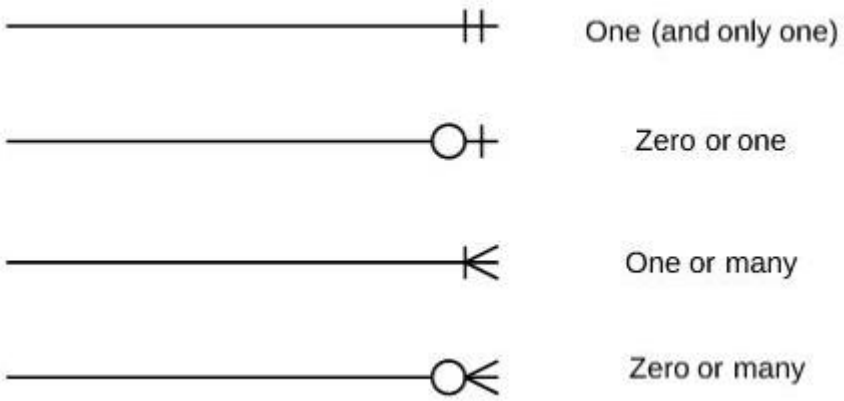
  

To implement a many-to-many relationship in a relational database, we usually decompose a many-to-many relationship into two (or more) one-to-many relationships.

e.g.



Other symbols used to describe relationships include:



Quick Check

Refer to the following normalised tables covered earlier.

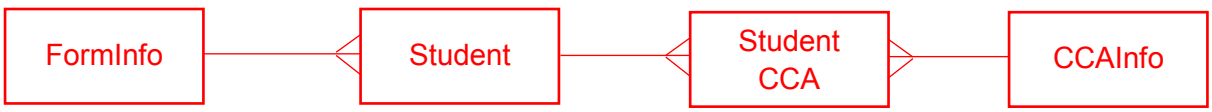
`Student(MatricNo, Name, Gender, FormClass)`

`FormInfo(FormClass, FormTutor, BaseClass)`

`StudentCCA(MatricNo, CCAName)`

`CCAInfo(CCAName, CCATeacherIC)`

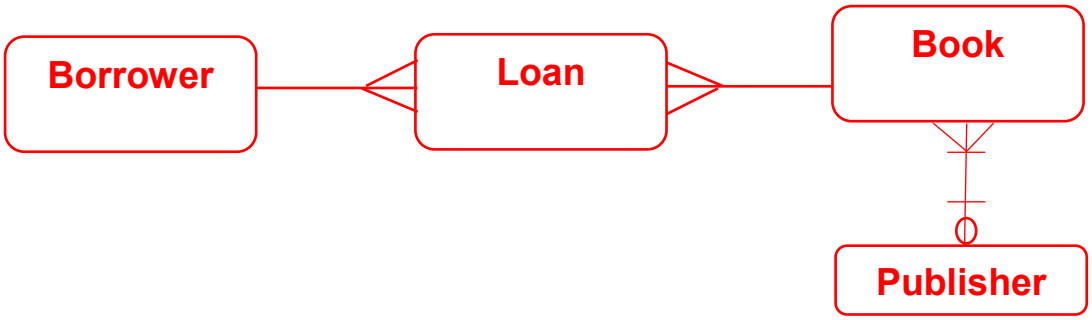
Draw an E-R diagram to model the simple school database described above.



Quick Check
A school library contains books that can be on loan to borrowers.

- A borrower can take one or more loans.
- Each loan record belongs to only one borrower.
- A book can be loaned many times.
- A publisher publishes one or more books.
- A book can be published by zero or one publisher.
  (e.g. exam papers and lecture notes are not published by an official publishing house.)

Draw an E-R diagram to model the school library database described above.



Structured Query Language (SQL)

Structured Query Language (SQL) is a standard computer language for the operation and management of relational databases. It is a language used to query, insert, update and modify data.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organisation for Standardisation (ISO) in 1987. Since then, the standard was updated several times. Most major relational databases support this standard, but have their own proprietary extensions.

There are many types of SQL database engines. A database engine is the software that a database management system (DBMS) uses to create, read, update and delete (CRUD) data from a database.

We are going to use **SQLite**, a widely used database engine, for the purpose of the syllabus. It is a popular choice as embedded database software for local/client storage in application software, such as web browsers.

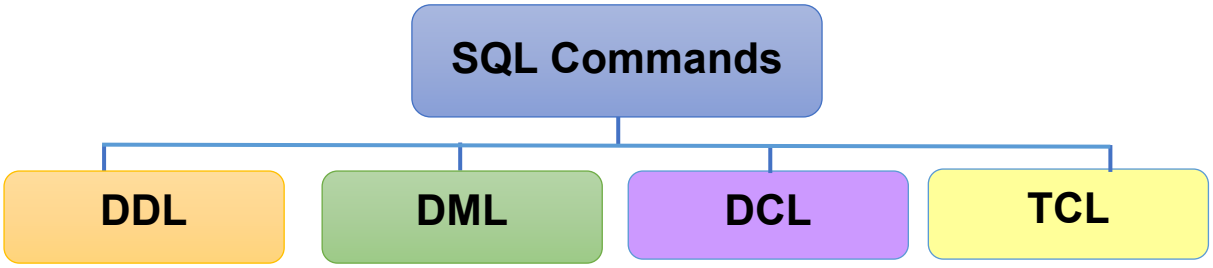
Python's IDLE comes with a built-in module for **SQLite3**.

To visualise the databases that we are going to encounter throughout the course of this study, we shall make use of **DB Browser for SQLite**.

Database Operations

In industry-based database applications, all four categories of SQL commands listed below are required.

- **Data Definition Language (DDL)** defines database schemas.
- **Data Manipulation Language (DML)** is used to retrieve and modify data.
- **Data Control Language (DCL)** is used to control access to a database.
- **Transaction Control Language (TCL)** is used to manage changes to a database, usually at transactional level.



CREATE **ALTER** **DROP** **RENAME** **TRUNCATE** **COMMENT**	**INSERT** **SELECT** **UPDATE** **DELETE** **MERGE** **CALL** **EXPLAIN PLAN** **LOCK TABLE**	**GRANT** **REVOKE**	**COMMIT** **SAVEPOINT** **ROLLBACK**

Some of the more advanced commands under DCL and TCL are more relevant to industry-specific roles, such as database administrators.

For the purpose of our learning, we only need to be able to understand and apply these basic CRUD database operations:

Operation	SQL Command
CREATE	INSERT
READ	SELECT
UPDATE	UPDATE
DELETE	DELETE

SQL Data Types

Each field in an SQL table has to be associated with one data type. The following table shows some of the common data types.

Data Type	SQL Syntax	Description
String	`CHAR(x)`	Fixed length characters (`x` can be from 1 to 255)
	`VARCHAR(x)`	Variable length characters (`x` can be from 1 to 65535)
	`TEXT`	Equivalent to `VARCHAR(65535)`
Numeric	`INTEGER`	Integers
	`REAL`	Real numbers
Boolean	`BOOL`	True or False

Creating and Manipulating SQL Database

Refer to the school library database that we have discussed earlier on Page 11.

Open `sql_lecture.db` in DB Browser for SQLite. Three tables - `Book`, `Publisher` and `Unused` (which shall be deleted later on), have been defined.

The summary of the tables required in this particular database, together with the fields and their constraints, are shown below.

`Borrower`

Field	Data Type	Constraint
BorrowerID	Numeric	PRIMARY KEY, AUTOINCREMENT
FirstName	String	NOT NULL
Surname	String	NOT NULL
ContactNum	Numeric	

`Loan`

Field	Data Type	Constraint
LoanID	Numeric	PRIMARY KEY, AUTOINCREMENT
BorrowerID	Numeric	FOREIGN KEY to `BorrowerID` in `Borrower` table
BookID	Numeric	FOREIGN KEY to `BookID` in `Book` table
DateBorrowed	String	(Desired format: YYYYMMDD)

`Book`

Field	Data Type	Constraint
BookID	Numeric	PRIMARY KEY, AUTOINCREMENT
BookTitle	String	NOT NULL
PublisherID	Numeric	FOREIGN KEY to `PublisherID` in `Publisher` table
Damaged	Numeric	NOT NULL (0 means undamaged, 1 means damaged)

`Publisher`

Field	Data Type	Constraint
PublisherID	Numeric	PRIMARY KEY, AUTOINCREMENT
PublisherName	String	NOT NULL

DDL: CREATE

The `CREATE` command allows us to make a new table.

```
CREATE TABLE <table_name> (
      <column1_name COLUMN1_TYPE COLUMN1_CONSTRAINT(S)>,
      <column2_name COLUMN2_TYPE COLUMN2_CONSTRAINT(S)>,
      …
      PRIMARY KEY (<column1_name>, <column2_name>, …),
      FOREIGN KEY (<column_name>) REFERENCES <table_name>(<column_name>)
)
```

The field constraints that we need to know are as follows:

- `PRIMARY KEY`
- `FOREIGN KEY … REFERENCES …`
- `NOT NULL`
  A value must be inserted into the field.
- `UNIQUE`
  No two records can repeat the same value within the field.
- `AUTOINCREMENT`
  The integer value is automatically given by the database when not specified (+1).

The following SQL statements, separated by a semi-colon, create the `Borrower` and `Loan` tables respectively in the database.

```
CREATE TABLE Borrower (
      BorrowerID INTEGER PRIMARY KEY AUTOINCREMENT,
      FirstName  VARCHAR(30) NOT NULL,
      Surname    VARCHAR(30) NOT NULL,
      Contact    INTEGER
);

CREATE TABLE Loan (
      LoanID           INTEGER PRIMARY KEY AUTOINCREMENT,
      BorrowerID       VARCHAR(30) NOT NULL,
      BookID           VARCHAR(30) NOT NULL,
      DateBorrowed     VARCHAR(30) NOT NULL
)
```

DDL: DROP

The `DROP` command allows us to delete an entire table and all the records inside.

```
DROP TABLE <table_name>
```

e.g. `DROP TABLE Unused`

DML: INSERT

The `INSERT` command allows us to insert a new record in a table.

```
INSERT INTO <table_name>(<column1_name, column2_name, …>)
VALUES (<column1_value, column2_value, …>)
```

Refer to the `Publisher` table below.

PublisherID	PublisherName
1	NPH
2	Unpop
3	Appleson
4	Squirrel
5	Yellow Flame

e.g.
```
INSERT INTO Publisher VALUES (6, 'BigBooks')
```
OR
```
INSERT INTO Publisher(PublisherName) VALUES ('BigBooks')
```

Either statement inserts a new publisher named `'BigBooks'` with `PublisherID = 6`. It is not necessary to specify `PublisherID` in this case since it is incremented automatically.

As a quick exercise, insert the following records into the `Borrower` and `Loan` tables.

`Borrower`

BorrowerID	FirstName	Surname	ContactNum
1	Peter	Tan	999
2	Sarah	Lee	81111123
3	Kumara	Ravi	94456677
4	Some	User	

`Loan`

LoanID	BorrowerID	BookID	DateBorrowed
1	3	2	20190220
2	3	1	20181215
3	2	3	20181231
4	1	5	20190111

DML: SELECT

The `SELECT` command allows us to retrieve data from the database.

```
SELECT <column1_name, column2_name, …>
FROM <table_name>
WHERE <condition(s)>
ORDER BY <column_name> ASC/DESC
```

Refer to the `Book` table below.

BookID	BookTitle	PublisherID	Damaged
1	The Lone Gatsby	5	0
2	A Winter's Slumber	4	1
3	Life of Pie	4	0
4	A Brief History of Primates	3	0
5	To Praise a Mocking Bird	2	0
6	The Catcher in the Eye	1	1
7	H2 Computing Ten Year Series		0

To select all fields from a table, we use `*`.

e.g. `SELECT * FROM Book`

To select only one or a subset of fields, we use the field names separated by commas.

e.g.
```
SELECT BookTitle FROM Book
SELECT BookID, BookTitle FROM Book
```

To select only rows meeting certain conditions, we use `WHERE`.

e.g. `SELECT BookTitle from Book WHERE Damaged = 1`
This statement returns the titles of all the damaged books.

`SELECT * from Book WHERE PublisherID IS NOT NULL`
This statement returns all the books with `PublisherID`.

`SELECT * from Book WHERE PublisherID = 4 AND Damaged = 0`
This statement returns all the books published by a certain publisher with ID no. 4 and are not damaged.

To order the selected records according to some field values in ascending or descending order, we use `ORDER BY … ASC/DESC`.

e.g. `SELECT BookID, BookTitle FROM Book ORDER BY PublisherID ASC`
This statement returns all the book IDs and titles arranged in an ascending order of `PublishedID`.

DML: UPDATE

The `UPDATE` command allows us to edit the data values in a database. One or more records may be updated at the same time.

```
UPDATE <table_name>
SET <column1_name = column1_value, column2_name = column2_value, …>
WHERE <condition(s)>
```

e.g.
```
UPDATE Book SET Damaged = 1
WHERE BookTitle = 'To Praise a Mocking Bird'
```
This statement updates the condition of the book titled 'To Praise a Mocking Bird' to damaged.

```
UPDATE Book SET BookTitle = 'Book: ' || Title
```
This statement updates the values of `BookTitle` such that each book title now starts with 'Book: '. Note the use of || for string concatenation.

DML: DELETE

The `DELETE` command allows us to delete existing records in a table.

```
DELETE FROM <table_name>
WHERE <condition(s)>
```

e.g.
```
DELETE FROM Publisher WHERE PublisherID = 6
```
This statement deletes the record having `PublisherID = 6`.

```
DELETE FROM Publisher
```
This statement deletes all the records in the `Publisher` table.

Quick Check

For the `Book` table, write an SQL statement to insert an undamaged book titled “Eleventh Night” with BookID no. 8 and PublisherID no. 2

```
INSERT INTO Book
VALUES (8, 'Eleventh Night', 2, 0)
```

For the `Book` table, write an SQL statement to update the condition of the book titled “Eleventh Night” to damaged.

```
UPDATE Book
SET Damaged = 1
WHERE BookTitle = 'Eleventh Night'
```

For the `Book` table, write an SQL statement to retrieve the titles of all the books with publishers and are damaged.

```
SELECT BookTitle FROM Book
WHERE PublisherID IS NOT NULL AND Damaged = 1
```

For the `Borrower` table, write an SQL statement to delete all the records without contact numbers.

```
DELETE FROM Borrower
WHERE ContactNum IS NULL
```

What is the difference between the two commands below?

```
DROP TABLE Table1

DELETE FROM Table2
```

`DROP TABLE` deletes the table and all the records inside. Since the table has been deleted, it is no longer possible to add records into `Table1` anymore.

`DELETE FROM` does not delete the table, but only all the records inside. That means it is possible to add records again into `Table2`.

JOIN

The `JOIN` command allows us to combine data from two tables.

Inner join returns the Cartesian product of rows from the tables, i.e. it combines each row in the first table with each row in the second table.

For example, to check the name of the publisher of each of the books in the library database, we can write the following SQL statement.

```
SELECT * FROM Book, Publisher
```

BookID	BookTitle	PublisherID	Damaged	PublisherID	PublisherName
1	The Lone Gatsby	5	0	1	NPH
1	The Lone Gatsby	5	0	2	Unpop
1	The Lone Gatsby	5	0	3	Appleson
1	The Lone Gatsby	5	0	4	Squirrel
1	The Lone Gatsby	5	0	5	Yellow Flame
2	A Winter's Slumber	4	1	1	NPH
2	A Winter's Slumber	4	1	2	Unpop
2	A Winter's Slumber	4	1	3	Appleson
2	A Winter's Slumber	4	1	4	Squirrel
2	A Winter's Slumber	4	1	5	Yellow Flame
3	Life of Pie	4	0	1	NPH
3	Life of Pie	4	0	2	Unpop
3	Life of Pie	4	0	3	Appleson
3	Life of Pie	4	0	4	Squirrel
3	Life of Pie	4	0	5	Yellow Flame
4	A Brief History Of Primates	3	0	1	NPH
4	A Brief History Of Primates	3	0	2	Unpop
4	A Brief History Of Primates	3	0	3	Appleson
4	A Brief History Of Primates	3	0	4	Squirrel
4	A Brief History Of Primates	3	0	5	Yellow Flame
5	To Praise a Mocking Bird	2	0	1	NPH
5	To Praise a Mocking Bird	2	0	2	Unpop
5	To Praise a Mocking Bird	2	0	3	Appleson
5	To Praise a Mocking Bird	2	0	4	Squirrel
5	To Praise a Mocking Bird	2	0	5	Yellow Flame
6	The Catcher in the Eye	1	1	1	NPH
6	The Catcher in the Eye	1	1	2	Unpop
6	The Catcher in the Eye	1	1	3	Appleson
6	The Catcher in the Eye	1	1	4	Squirrel
6	The Catcher in the Eye	1	1	5	Yellow Flame
7	H2 Computing Ten Year Series		0	1	NPH
7	H2 Computing Ten Year Series		0	2	Unpop
7	H2 Computing Ten Year Series		0	3	Appleson
7	H2 Computing Ten Year Series		0	4	Squirrel
7	H2 Computing Ten Year Series		0	5	Yellow Flame

The resulting table is a big table having many records with inconsistent data for `PublisherID`. In order to retrieve only the useful records, we can add a condition as follows.

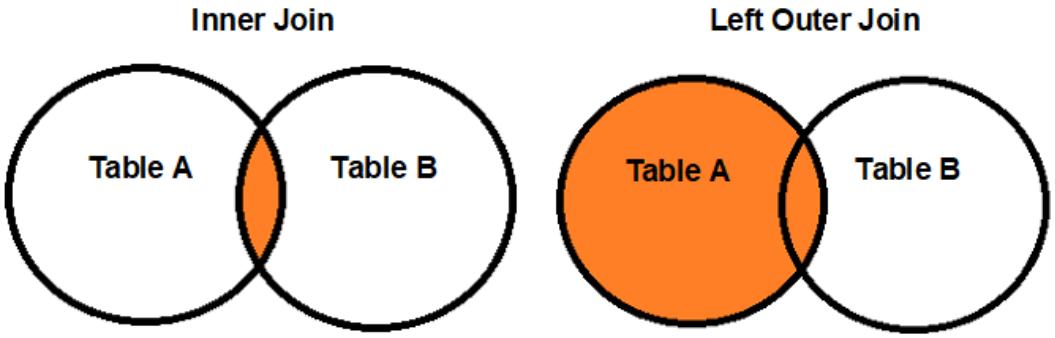
```
SELECT * FROM Book, Publisher
WHERE Book.PublisherID = Publisher.PublisherID
```

BookID	BookTitle	PublisherID	Damaged	PublisherID	PublisherName
1	The Lone Gatsby	5	0	5	Yellow Flame
2	A Winter's Slumber	4	1	4	Squirrel
3	Life of Pie	4	0	4	Squirrel
4	A Brief History Of Primates	3	0	3	Appleson
5	To Praise a Mocking Bird	2	0	2	Unpop
6	The Catcher in the Eye	1	1	1	NPH

The table above is more meaningful as it links the book titles to the correct publishers. However, notice that H2 Computing Ten Year Series has been omitted as it has no `PublisherID`.

In such a case, we need to use **left outer join**, which takes into consideration all the records from one table and records from the other that meet the join conditions.

```
SELECT <column1_name, column2_name, …>
FROM <Table_A>
INNER / LEFT OUTER JOIN <Table_B>
ON <condition(s)>
```




```
SELECT * FROM Book
LEFT OUTER JOIN Publisher
ON Book.PublisherID = Publisher.PublisherID
```

BookID	BookTitle	PublisherID	Damaged	PublisherID	PublisherName
1	The Lone Gatsby	5	0	5	Yellow Flame
2	A Winter's Slumber	4	1	4	Squirrel
3	Life of Pie	4	0	4	Squirrel
4	A Brief History Of Primates	3	0	3	Appleson
5	To Praise a Mocking Bird	2	0	2	Unpop
6	The Catcher in the Eye	1	1	1	NPH
7	H2 Computing Ten Year Series		0		

Quick Check
Write an SQL statement to retrieve the titles of all the books that are not damaged with their publisher names.

```
SELECT BookTitle, PublisherName FROM Book, Publisher
WHERE Book.PublisherID = Publisher.PublisherID AND Book.Damaged = 0
```

AGGREGATE FUNCTIONS

There are a few aggregate functions that we can use in SQL statements to calculate results from a given database:

- `MIN` (minimum value)
- `MAX` (maximum value)
- `SUM` (sum of all values)
- `COUNT` (number of values)

OPERATORS

We have seen some operators being used in the examples earlier. These operators are often used in the `SELECT` statements, but can be used in other statements like `UPDATE`. The following are the three types of operators that we are expected to know.

Comparison Operators

=	<	>
!=	<=	>=

Logical Operators

OR	IS	||
AND	IS NOT	(string concatenation)

Arithmetic Operators

+	*	%
-	/	

Python and SQLite

DB Browser for SQLite is a convenient program for us to experiment with SQL statements and examine the results. However, it is not an appropriate program to use if we want to customise or restrict how the contents of a database are modified or presented.

Suppose we have a database that stores information about the books in a library. We should not use DB Browser for SQLite for users to search the database as not everyone is familiar with SQL statements. That aside, malicious users may run harmful statements, e.g. `DROP TABLE` to delete the database.

As such, a developer typically write a custom program to control how users interact with a database, which has an interface that is easy to understand and use. Based on the users' inputs, the program would then generate the appropriate SQL statements in the background and run them to produce the intended results. In this way, the users are prevented from modifying the database.

We shall learn how to write Python programs that can interact with SQLite databases using the built-in `sqlite3` module.

Quick Check
Which of the following is not a valid reason why DB Browser for SQLite should not be accessible to the users of a public library?

A Users may use the program to insert fake data into the database.

B Users may use the program to drop tables from the database.

C Users may use the program to perform a query that returns nothing.

D Users may not know how to perform the query using the program.

Loading a Database

Program 1: `load_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.close()
```

The `connect()` method (line 3) takes in a string that contains the path and filename of a database file and returns a `Connection` object. If no path is included, the file is assumed to be in the same directory as the Python file. Furthermore, if the specified file does not exist, an empty file will be created with the given filename instead.

After all operations with the database are complete, the `close()` method (line 4) of the `Connection` object should then be called. This ensures that the database file is closed properly, but does not save any modifications that have been made to the data.

Executing SQL Statements

Program 2: `create_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.execute("CREATE TABLE Book " +
5                     "(ID INTEGER PRIMARY KEY, Title TEXT)")
6  connection.commit()
7  connection.close()
```

The `execute()` method (line 4) takes in a string containing the SQL statement we wish to run.

The `commit()` method (line 6) saves the change(s) made to the database.

After running the program above, we can use DB Browser for SQLite to check that a table called `Book` has indeed been created.

However, if we try to run the program again, we will get the following error:

```
Traceback (most recent call last):
  File "create_example.py", line 5, in <module>
    "(ID INTEGER PRIMARY KEY, Title TEXT)")
sqlite3.OperationalError: table Book already exists
```

This demonstrates that calling `execute()` is just like running regular SQL statements in the "Execute SQL" tab of DB Browser for SQLite. Any errors caused by running SQL statements are reported as Python exceptions.

Committing Changes and Rolling Back

Program 3: `insert_example_incomplete.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.execute("INSERT INTO Book(ID, Title) " +
5                     "VALUES(0, 'Example Book')")
6  connection.close()
```

The program above runs with no errors. However, if we open `library.db` using DB Browser for SQLite, we can see that the inserted data is missing from the `Book` table.

A **transaction** is a unit of work that is performed against a database. Using `INSERT`, `UPDATE` or `DELETE` command opens a transaction that can either be **committed** or **rolled back**.

Program 4: `insert_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.execute("INSERT INTO Book(ID, Title) " +
5                     "VALUES(0, 'Example Book')")
6  connection.commit()
7  connection.close()
```

With a call to `commit()` added on line 6, the data are inserted and saved correctly.

Program 5: `rollback_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4
5  connection.execute("INSERT INTO Book(ID, Title) " +
6                     "VALUES(1, 'Rollback Book')")
7  connection.execute("INSERT INTO Book(ID, Title) " +
8                     "VALUES(2, 'Also Rollback Book')")
9  connection.rollback()
10
11 connection.execute("INSERT INTO Book(ID, Title) " +
12                    "VALUES(3, 'Committed Book')")
13 connection.commit()
14
15 connection.close()
```

The `rollback()` method (line 9) discards any changes done by the preceding SQL statements. In the example shown above, the first two `INSERT` statements are rolled back so that they have no effect on the database. On the other hand, the last `INSERT` statement is committed so it does affect the database.

This behaviour of SQLite is useful as sometimes we may wish to discard any modifications since the last transaction was opened. For instance, in our library example, we may start the process of placing a book on loan, but discover partway that the borrower has already reached his limit of borrowed books. We can discard all the changes made since the transaction was opened by calling the `Connection` object's `rollback()` method.

Warning: Starting with Python 3.6, commands that control the structure of the database, such as `CREATE TABLE` and `DROP TABLE`, do not open a transaction and will generally take effect immediately. This means that, by default, it is not possible to roll back such changes automatically.

Parameter Substitution

Program 6: `delete_example.py`

```
1   import sqlite3
2
3   connection = sqlite3.connect("library.db")
4
5   # Insert some rows first so we have something to delete
6   connection.execute("INSERT INTO Book(ID, Title) " +
7                      "VALUES(4, 'Extra Book')")
8   connection.execute("INSERT INTO Book(ID, Title) " +
9                      "VALUES(5, 'Also Extra Book')")
10  connection.commit()
11
12  # Ask for ID and delete the corresponding row
13  book_id = input("Enter Book ID to delete: ")
14  connection.execute("DELETE FROM Book WHERE ID = ?", (book_id,))
15  connection.commit()
16
17  connection.close()
```

We often need to include some data that are provided by the user. For instance, we may want the user to enter the ID of a book to delete from the database. This requires us to generate a `DELETE` statement with the entered ID in its `WHERE` clause.

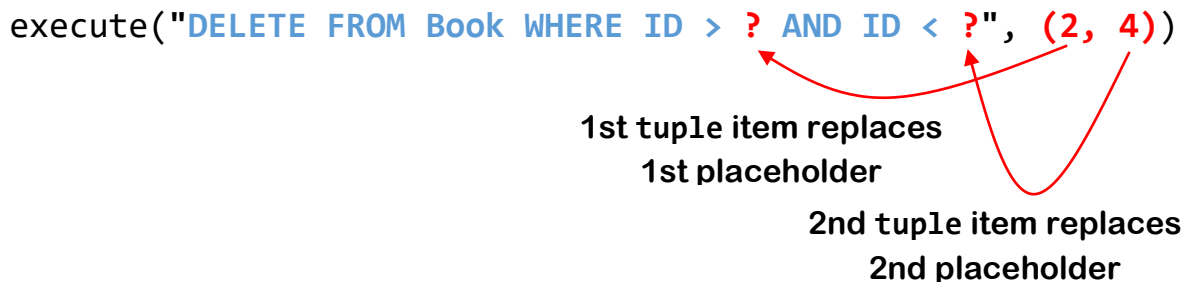
We may be tempted to use string concatenation to generate the required SQL statement,

e.g. `connection.execute("DELETE FROM Book WHERE ID = " + book_id)`

Unfortunately, this is insecure as special characters or keywords in the user's input are not escaped, thus malicious users can use this loophole to inject his own SQL statements.

We should use **parameter substitution** to safely include data that is provided by the user. To do this, we use the question-mark character `?` as placeholders for any data provided by the user. We then provide a second argument to `execute()` that is a `tuple` of values to fill in the placeholders.

Parameter substitution follows the same order in which the placeholders appear in the SQL statement. This is illustrated by the following diagram:



Quick Check
As mentioned previously, the following string concatenation is not safe.

```
connection.execute("DELETE FROM Book WHERE ID = " + book_id)
```

Suggest an input for `book_id` that will delete all the rows in the `Book` table.

`1 or 1`

Retrieving Data from a Database

As we have already learned, the `SELECT` command is used to select data from the database. When we run a `SELECT` command in DB Browser for SQLite, the selected rows are usually displayed in a table.

In Python, however, we must access the selected rows using a `Cursor` object that is returned by the `execute()` method. This cursor can go through the selected rows, one by one, using either a `for` loop or the `fetchone()` method. Each iteration returns a `tuple` of the columns in the current row.

The two programs below print out all the book titles in the `Book` table.

Program 7: `forloop_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  cursor = connection.execute("SELECT ID, Title FROM Book")
5  for row in cursor:
6      print(row[1])    # Title is the second item in the tuple
7  connection.close()
```

Program 8: `fetchone_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  cursor = connection.execute("SELECT ID, Title FROM Book")
5  row = cursor.fetchone()
6  while row is not None:
7      print(row[1])    # Title is the second item in the tuple
8      row = cursor.fetchone()
9  connection.close()
```

The `fetchone()` method (Program 8 line 5) will advance the cursor to the next row, so calling it repeatedly will iterate through the selected rows until the cursor reaches the end and returns `None`.

Program 9: `fetchall_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  cursor = connection.execute("SELECT ID, Title FROM Book")
5  rows = cursor.fetchall()
6  for row in rows:
7      print(row[1])    # Title is the second item in the tuple
8  connection.close()
```

Alternatively, instead of going through the rows one by one using a cursor, we may wish to fetch all the rows at once and keep them in a `list`.

The `fetchall()` method (line 5) returns a `list` of `tuples` with each `tuple` containing the selected columns for a single row.

Program 10: `row_factory_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.row_factory = sqlite3.Row
5  cursor = connection.execute("SELECT ID, Title FROM Book")
6  for row in cursor:
7      print(row["Title"])    # row is now a dictionary
8  connection.close()
```

Yet another alternative is to configure the SQLite connection so that each row is retrieved as a `dictionary` that maps column names to field values instead. To do this, we set the connection object's `row_factory` attribute to the built-in `sqlite3.Row` class (line 4). This lets us change the ordering of columns in the `SELECT` statement without having to modify the code for extracting individual column values.

Quick Check

Refer to Program 10.

The SQL statement on line 5 is replaced with one of the following options. Which option would cause an error on line 7 when the program is run?

A `SELECT * FROM Book`

B `SELECT ID FROM Book`

C `SELECT Title FROM Book`

D `SELECT Title, ID FROM Book`

`sqlite3` Module Summary

`connect(filename)`	Creates a `Connection` object using SQLite file with the given `filename`
`Row`	Can be used as a `Connection` object's `row_factory` so that `fetchone()` returns a `dictionary` that maps column names to field values instead of returning a `tuple` of values

`Connection` Class Summary

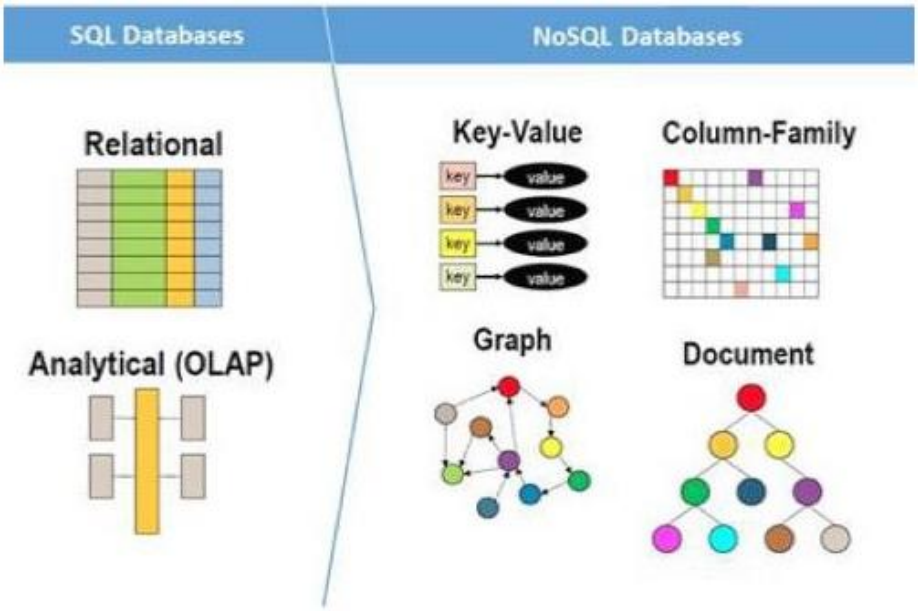
`commit()`	Saves changes to (but does not close) SQLite file
`close()`	Closes (but does not save changes to) SQLite file
`execute(sql)`	Runs the given SQL statement on the database and returns a `Cursor` object
`execute(sql, values_tuple)`	Runs the given SQL statement (first argument) after substituting question mark(s) with the corresponding value(s) in the given `tuple` (second argument) and returns a `Cursor` object
`rollback()`	Undoes any changes made since the last call to `commit()`
`row_factory`	Can be set to `Row` so that `fetchone()` returns a `dictionary` that maps column names to field values instead of returning a `tuple` of values

`Cursor` Class Summary

`fetchone()`	Returns a `tuple` of values from next row of the query result or `None` if there are no more values (or a `dictionary` that maps column names to field values if `row_factory` is set to `Row`)
`fetchall()`	Calls `fetchone()` repeatedly until it returns `None` and returns a `list` of the non-`None` results

2021 JC2 H2 Computing 9569
25. Non-Relational Database: MongoDB

Introduction



In the previous chapter, we have learnt about relational (SQL) database involving fixed schema, which works well with structured data. However, with the increasing number of ways to generate and gather data, we often need to deal with unstructured data.

A **non-relational (NoSQL) database** uses a storage model optimised for the specific requirements of the types of data being stored in it instead of using tabular schema of rows and columns found in a relational database.

For the purpose of the syllabus, we shall focus on **MongoDB**, a type of document database, which deals with **JSON** (JavaScript Object Notation) documents.

Recall that in a hash table, each key points to a single value or data item. Python `dictionary` is implemented using a hash table, so that the values stored can be accessed directly by hashing the relevant keys. A **document** in MongoDB is akin to `dictionary`.

Here are the terms used in MongoDB with the corresponding terms in SQL for comparison.

MongoDB Term	SQL Term
Database	Database
Collection	Table
Document	Row
Field	Field / Column

SQL VS NoSQL Databases

SQL	NoSQL
Has fixed, predefined schema	No predefined schema, thus dynamic and can change easily
Data are stored in tables with a fixed data type in each field	Data are stored as collections of documents with no fixed data types
Joins are used to get data across tables, thus easier to use for complex queries	No join operations

The choice of database to use depends on the types of data being stored, as well as the nature of the tasks that the database is required to perform.

SQL databases should be used if:

- the data stored has a fixed schema with the atomicity, consistency, isolation and durability (ACID) properties critical to the database
- complex and varied queries will be frequently performed
- a high number of simultaneous transactions will be performed

Atomicity	A transaction takes place completely or does not happen at all.
Consistency	Integrity constraints are maintained at all times.
Isolation	Multiple transactions can occur concurrently without leading to the inconsistency of database state.
Durability	Once a transaction has been completed, the updates and modifications to the database are saved even if the system fails or restarts.

NoSQL databases should be used if:

- the data stored has a dynamic schema, i.e. unstructured data with flexible data types
- data storage needs to be performed quickly
- simple queries are often made due to better performance speed
- there will be an extremely large amount of data, i.e. big data

NoSQL databases address the shortcomings of SQL databases as follows:

- SQL databases have predefined schemas that are difficult to change. Should we wish to add a field to only a small number of records, we need to include the field for the entire table.

  Therefore, it can be difficult to support the processing of unstructured data using SQL databases, unlike in NoSQL databases where data are stored in documents that need not be of the same format.

- Unlike NoSQL databases, relational databases do not usually support **hierarchical data storage** where less frequently used data are moved to cheaper, slower storage devices.

  This means that the cost of storing the same amount of data in an SQL database is more expensive than in a NoSQL database.

- An SQL database is stored in one server, which makes the database unavailable when the server fails.

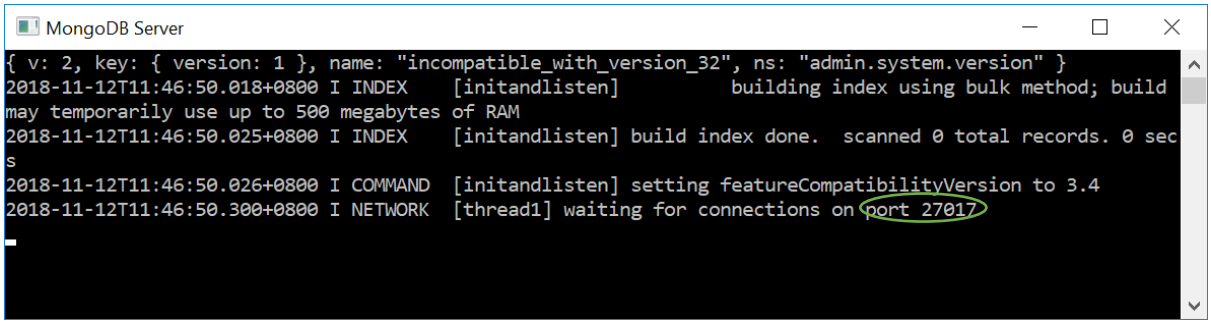
  NoSQL databases, on the other hand, are designed to take advantage of multiple servers so that if one server fails, the other servers can continue to support applications.

- SQL databases are mainly **vertically scalable**, which means that improving the performance usually requires upgrading the existing server with faster processors and more memory space. Such high-performance components can be expensive and upgrades are limited by the capacity of a single machine.

  On the other hand, NoSQL databases are **horizontally scalable**, which means that the performance can be improved by simply increasing the number of servers. This is relatively cheaper as mass-produced average-performance computers are easily available at low prices.

Python and MongoDB

To interact with MongoDB databases, we need to first connect to the MongoDB server. The server window should remain open as long as we are accessing the database.



Just like how Python can interact with SQLite databases, it can also do the same with MongoDB databases. For the purpose of the latter, we use the built-in `pymongo` module.

Connecting to a Database

Program 1: `access.py`

```
1  import pymongo
2  client = pymongo.MongoClient("127.0.0.1", 27017)
3  databases = client.database_names()
4
5  print("The databases in the MongoDB server are:")
6  print(databases)
7
8  client.close()
```

The `MongoClient()` method (line 3) connects to the local MongoDB database, which is at port 27017 by default. The port number can be seen when we start the MongoDB server. The IP address 127.0.0.1 is the localhost IP address.

The `database_names()` method (line 4) retrieves the names of the databases as a `list`.

The `close()` method (line 9) closes the connection to the server.

Inserting Documents

Program 2: `insert.py`

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("movies")
6
7   coll.insert_one({"_id":1, "title":"Johnny Math",
    "genre":"comedy"})
8   coll.insert_one({"title":"Star Walls", "genre":"science
    fiction"})
9   coll.insert_one({"title":"Detection"}) #no genre
10
11  list_to_add = []
12  list_to_add.append({"title":"Badman", "genre":"adventure",
    "year":2015})
13  list_to_add.append({"title":"Averages", "genre":["science
    fiction","adventure"], "year":2017})
14  list_to_add.append({"title":"Octopus Man",
    "genre":"adventure", "year":2017})
15  list_to_add.append({"title":"Fantastic Bees",
    "genre":"adventure", "year":2018})
16  list_to_add.append({"title":"Underground", "genre":"horror",
    "year":2014})
17  coll.insert_many(list_to_add)
18
19  c = db.collection_names("entertainment")
20  print("Collections in entertainment database: ", c)
21
22  client.close()
```

Program 2 above demonstrates two ways of inserting documents.

The `get_database()` method (line 4) and `get_collection()` method (line 5) allows us to access a specific database and a collection respectively. They are created when not already available in the server.

The `insert_one()` method (lines 7-9) allows the insertion of one document at a time.

The `insert_many()` method (line 17) allows the insertion of multiple documents in a `list`.

Note that MongoDB assigns a unique `_id` to each document inserted. The value of `_id` can be customised during the insertion process (line 7), but in so doing, we cannot run Program 2 again until we remove this document. Otherwise, the program will produce an error as we cannot have more than one document of the same `_id`. When we run Program 2 again with line 7 commented out, duplicates of the other documents will be created.

It is possible to write a program to read data from a delimited text file and insert the documents into the database. An input file (with each row containing the name and the age of a user) and parts of the program are given below.

Input File: `input.txt`

```
Amanda,45
Bala,28
Charlie,33
Devi,29
```

Fill in the blanks below.

Program 3: `insert_from_txt.py`

```
1   import pymongo, csv
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("users")
6
7   with open('input.txt') as csv_file:
8       csv_reader = csv.reader(csv_file, delimiter=',')
9       for row in csv_reader:
10          coll.insert_one({"name":row[0], "age":int(row[1])})
11
12  client.close()
```

Retrieving Documents

The following is a list of commonly used query operators.

<table>
<tr><td>$eq</td><td>Equal to</td><td rowspan="6">{'<i>field</i>': {'$<i>op</i>': …}}</td></tr>
<tr><td>$ne</td><td>Not equal to</td></tr>
<tr><td>$gt</td><td>Greater than</td></tr>
<tr><td>$gte</td><td>Greater than or equal to</td></tr>
<tr><td>$lt</td><td>Less than</td></tr>
<tr><td>$lte</td><td>Less than or equal to</td></tr>
<tr><td>$or</td><td colspan="2">Logical OR

{'$or': [{'field1': …}, {'field2': …}, …]}</td></tr>
<tr><td>$and</td><td colspan="2">Logical AND

{'$and': [{'field1': …}, {'field2': …}, …]}</td></tr>
<tr><td>$not</td><td colspan="2">Logical NOT (also retrieve documents that do not have field)

{'field': {'$not': {…}}}</td></tr>
<tr><td>$exists</td><td colspan="2">Retrieve documents that have the named field

{'field': {'$exists': True/False}}</td></tr>
<tr><td>$in</td><td colspan="2">Retrieve documents that have at least one of the items in the list

{'field': {'$in': ['item1', 'item2', …]}}</td></tr>
<tr><td>$nin</td><td colspan="2">Retrieve documents that do not have at least one of the items in the list

{'field': {'$nin': ['item1', 'item2', …]}}</td></tr>
</table>

Program 4: `view1.py`

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("movies")
6
7   result = coll.find()
8   print("All documents in movies collection: ")
9   for document in result:
10      print(document)
11  print("Document count in movies collection:", result.count())
12  print()
13
14  query = {'genre':'adventure','year': {'$gt': 2016}}
15  result = coll.find(query)
16  print("All movie titles with adventure genre after 2016:")
17  for document in result:
18      print(document['title'])
19  print("There are", result.count(), "such movies.")
20  print()
21
22  result = coll.find_one({'genre':'adventure'})
23  print("One movie with adventure genre:", result)
24
25  client.close()
```

The `find()` method (lines 7 and 15) returns a `Cursor` of the documents in the `movies` collection. When a query is not supplied as an argument, it returns all the documents available in the collection. The result can then be printed by means of a loop. Each document is in the form of `dictionary` in Python.

The `find_one()` method (line 22) retrieves only one document according to the order of insertion into the collection.

The `count()` method (lines 11 and 19) returns the number of documents.

Program 5: `view2.py`

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("movies")
6
7   result = coll.find({'genre':{'$in':['adventure','comedy']}})
8   print("All movies with adventure or comedy genre:")
9   for document in result:
10      print(document)
11  print()
12
13  query = {'genre':{'$exists':False}}
14  result = coll.find(query)
15  print("All movies without genre:")
16  for document in result:
17      print(document['title'])
18
19  client.close()
```

Quick Check

Modify the program above to print out:

- all movies without 'adventure' and 'comedy' as their genres in lines 7-11
- the movie title and how many years ago was the movie released for all movies with year given in lines 13-17

```
result = coll.find({'genre':{'$nin':['adventure', 'comedy']}})
print("All movies without adventure and comedy genres:")

###

query = {'year': {'$exists':True}}
result = coll.find(query)
print("All movies with year given:")
for document in result:
      age = 2021 - document['year']
      print(" - Title: ", document['title'],
            ", no. of year(s) since release: ", age)
```

Updating Documents

Program 6: update.py

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("movies")
6
7   result = coll.find()
8   print("All documents in movies collection:")
9   for document in result:
10      print(document)
11  print()
12
13  search = {'year':{'$gt':2016}}
14  update = {'$set':{'year':2015}}
15  coll.update_one(search, update)
16
17  result = coll.find()
18  print("All documents in movies collection after 1st update:")
19  for document in result:
20      print(document)
21  print()
22
23  search = {'year':{'$eq':2015}}
24  update = {'$unset':{'year':0}}
25  coll.update_many(search, update)
26
27  result = coll.find()
28  print("All documents in movies collection after 2nd update:")
29  for document in result:
30      print(document)
31  print()
32
33  client.close()
```

The `$set` operator (line 13) is called to edit the value(s) of key(s).

The `update_one()` method (line 15) updates the first document that matches the given criteria.

The `$unset` operator (line 24) is called to remove key-value pair(s).

The `update_many()` method (line 25) updates all the documents that match the given criteria.

Deleting Documents

Program 7: `delete.py`

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("movies")
6
7   result = coll.count()
8   print("Document count in movies collection:", result)
9
10  coll.delete_one({'year':2017})
11
12  result = coll.count()
13  print("Document count in movies collection after 1st deletion
    one:", result)
14
15  coll.delete_many({'year':{'$exists':'false'}})
16
17  result = coll.count()
18  print("Document count in movies collection after 2nd deletion
    one:", result)
19
20  client.close()
```

The `delete_one()` method (line 10) deletes the first document that matches the given criteria.

The `delete_many()` method (line 15) deletes all the documents that match the given criteria.

Dropping a Collection

Program 8: drop_collection.py

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5   coll = db.get_collection("movies")
6
7   result = coll.count()
8   print("Document count in movies collection:", result)
9
10  db.drop_collection("movies")
11
12  result = coll.count()
13  print("Document count in movies collection after dropping:",
    result)
14
15  client.close()
```

The `drop_collection()` method (line 10) removes the collection with all the documents inside it.

Dropping a Database

Program 9: drop_database.py

```
1   import pymongo
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4   db = client.get_database("entertainment")
5
6   client.drop_database("entertainment")
7
8   databases = client.database_names()
9   print("The databases in the MongoDB server are:", databases)
10
11  client.close()
```

The `drop_database()` (line 6) method removes the entire database with all the collections and the documents inside it.

`pymongo` Module Summary

`MongoClient(IP, port)`	Creates a MongoDB `Client` object via connection to the given `IP` address and `port` number

`Client` Class Summary

`database_names()`	Shows all the databases in a `list`
`get_database(name)`	Declares a `Database` object of the given `name`
`drop_database(name)`	Deletes a `Database` object of the given `name`
`close()`	Closes the connection to MongoDB

`Database` Class Summary

`collection_names()`	Shows all the collections in a `list`
`get_collection(name)`	Declares a `Collection` object of the given `name`
`drop_collection(name)`	Deletes a `Collection` object of the given `name`

`Collection` Class Summary

`insert_one(nosql)`	Inserts one document given the `nosql` statement
`insert_many(list_of_nosql)`	Inserts multiple documents given the `list_of_nosql` statements
`find()`	Returns a `Cursor` of documents
`find_one(nosql)`	Returns the first document that matches the given `nosql` statement
`update_one(nosql1, nosql2)`	Finds the first document that matches the given `nosql1` statement and updates it given the `nosql2` statement
`update_many(nosql1, nosql2)`	Finds all documents that match the given `nosql1` statement and updates them given the `nosql2` statement
`delete_one(nosql)`	Deletes the first document that matches the given `nosql` statement
`delete_many(nosql)`	Deletes all documents that match the given `nosql` statement
`count()`	Returns the number of documents

2021 JC2 H2 Computing 9569
26. Data Management and Privacy

Introduction

Data management is an administrative process that involves acquiring, validating, storing, protecting and processing required data to ensure their integrity, accessibility and privacy for the users.

The value of physical equipment is often far less than that of the data it contains. The loss of data can be costly, even more so if they fall into the hands of unauthorised individuals. With more data handled across the globe, data protection and privacy is now more crucial than ever.

Backup and Archive

Backup and **archive** are two terms that are often mentioned in the same breath. On the surface, they may seem almost analogous, but are not the same. The table below highlights the key differences between the two.

	Backup	**Archive**
Nature of data	Live data that are frequently overwritten e.g. Drafts of newspaper articles before publication are crucial should the computer encounter an error.	Data that will not be subjected to any more changes e.g. Old newspaper articles, going as far back as the very first publication.
Data retention	Short-term e.g. Drafts of newspaper articles can be deleted as soon as the final version is published.	Long-term e.g. Old newspaper articles are retained indefinitely for future reference.
Retrieval speed	Should be fast e.g. Since the author of a newspaper article is still working on it, retrieval of previous drafts should not take long to restore should an error happen.	Can be slow e.g. Since old newspaper articles are not likely to be used as frequently as the recent ones, they could be stored in an inconvenient location. Preservation of the information is more important than making it readily accessible.

Version Control and Naming Convention

Version control is the practice of tracking and managing changes to data.

More often than not, several people are involved in a project over an extended period of time. Without proper controls, this can quickly lead to confusion as to which version is the most recent.

The usefulness of version control is as follows:

- Along with proper documentation, it allows for tracking of changes made to a particular project. A version control table can be maintained, noting the changes and their dates.

  For instance, in the case of a software, the members of the development team will have a clear idea of what are the features that have been implemented and others that still require implementation.

- It provides an efficient mechanism for backup with the ability to roll back to previous versions. Should serious issues be discovered in the current version, developers can roll back to the previous functioning version.

Naming convention should be established along with version control. For file names to be meaningful and easily retrievable, they have to be consistent with agreed vocabulary, numbering, punctuation, date format, etc. in a specific order.

As a simple example:

- Any major changes to a file can be indicated by:
  'v01' refers to the first version,
  'v02' refers to the second version, etc.

- Any minor changes can be indicated by:
  'v01.01' refers to the first minor change made to the first version
  'v03.02' refers to the second minor change made to the third version, etc.

Data Privacy

Data privacy is the requirement for data to be accessed by or disclosed to authorised individuals only. It is crucial that unauthorised people do not have access to data they are not supposed to have. Unfortunately, in today's digitised society, it is easier than ever to gather someone else's personal data.

For instance, data on which websites you frequently visit can reveal which products you are more likely to purchase as a shopper. This information can be highly valuable to an advertiser. As more services become available online, the risk of fraudulent use of data increases. Nowadays, with a photo of your identity card, a person can impersonate you and register for a new phone line on a telco website. Previously such a transaction would have required the person to personally register over the counter with the physical identity card. As technology becomes increasingly more powerful, machines can gather information on a person easily, like performing facial recognition on surveillance videos to track down the whereabouts of an individual in a particular area.

Personal Data Protection Act

In Singapore, personal data[1] is protected under the **Personal Data Protection Act (PDPA)**, a law comprising various rules that govern the collection, use, disclosure and care of personal data. It recognises both the rights of individuals to protect their personal data, including rights of access and correction, as well as the needs of organisations to collect, use or disclose personal data for legitimate and reasonable purposes.

It takes into account the following:

- **Consent** – Organisations must obtain an individual's knowledge and consent to collect, use or disclose his/her personal data (with some exceptions).

- **Notification** – Organisations must inform individuals of the purposes for collecting, using or disclosing their personal data.

- **Appropriateness** – Organisations may collect, use or disclose personal data only for purposes that would be considered appropriate to a reasonable person under the given circumstances.

- **Accountability** – Organisations must make information about their personal data protection policies available on request. They should also make available the business contact information of the representatives responsible for answering questions relating to the organisations' collection, use or disclosure of personal data.

To administer and enforce the PDPA, the government set up the **Personal Data Protection Commission (PDPC)** in 2013.

[1] Personal data refers to data, whether true or not, about an individual who can be identified from that data; or from that data and other information to which the organisation has or is likely to have access.

Organisations are required to abide by the following main personal data obligations:

1. Consent Obligation
Only collect, use or disclose personal data for purposes for which an individual has given his or her consent.

2. Purpose Limitation Obligation
An organisation may collect, use or disclose personal data about an individual for the purposes that a reasonable person would consider appropriate in the circumstances and for which the individual has given consent.

3. Notification Obligation
Notify individuals of the purposes for which your organisation is intending to collect, use or disclose their personal data on or before such collection, use or disclosure of personal data.

4. Access and Correction Obligation
Upon request, the personal data of an individual and information about the ways in which his or her personal data has been or may have been used or disclosed within a year before the request should be provided. However, organisations are prohibited from providing an individual access if the provision of the personal data or other information could reasonably be expected to cause harmful effects. Organisations are also required to correct any error or omission in an individual's personal data that is raised by the individual.

5. Accuracy Obligation
Make reasonable effort to ensure that personal data collected by or on behalf of your organisation is accurate and complete, if it is likely to be used to make a decision that affects the individual, or if it is likely to be disclosed to another organisation.

6. Protection Obligation
Make reasonable security arrangements to protect the personal data that your organisation possesses or controls to prevent unauthorised access, collection, use, disclosure or similar risks.

7. Retention Limitation Obligation
Cease retention of personal data or remove the means by which the personal data can be associated with particular individuals when it is no longer necessary for any business or legal purpose.

8. Transfer Limitation Obligation
Transfer personal data to another country only according to the requirements prescribed under the regulations, to ensure that the standard of protection provided to the personal data so transferred will be comparable to the protection under the PDPA, unless exempted by the PDPC.

9. Accountability Obligation
Make information about your data protection policies, practices and complaints process available on request. Designate a Data Protection Officer to ensure that your organisation complies with the PDPA.

More information on PDPA are available at the PDPC website: http://www.pdpc.gov.sg/

Example: Use of NRIC/FIN

The **Singapore National Registration Identification Card (NRIC)** number is a unique identifier assigned to Singapore citizens and permanent residents. Similarly, the **Foreign Identification Number (FIN)** is a unique identifier that is assigned to foreigners living in Singapore. The NRIC/FIN contains personal information about the individual, such as his/her date of birth and residential address. As unique identifiers like NRIC and FIN are permanent, irreplaceable and often used in a variety of government transactions, we need to be careful with such data.

Individuals should not readily provide their NRIC/FIN and personal particulars to other people or companies. Consent is required before organisations can obtain a person's data. Under the PDPA, from 1 September 2019, organisations[2] are generally not allowed to collect, use or disclose NRIC numbers (or copies of NRIC), except in the following circumstances:

- Collection, use or disclosure of NRIC numbers (or copies of NRIC) is required under the law (or an exception under the PDPA applies), or

- Collection, use or disclosure of NRIC numbers (or copies of NRIC) is necessary to accurately establish or verify the identities of the individuals to a high degree of fidelity.

For example, a medical clinic needs to see the NRIC of a patient to identify the person. The clinic will need to keep the NRIC number, name, residential address and contact number of the person with the medical notes for future reference. The PDPA allows for that. However, a shopping mall cannot collect the photographs of NRICs of all the shoppers that want to participate in their lucky draw. It is unnecessary to collect the photographs to verify the lucky draw participant. Instead, the participants can be identified with their mobile number or be asked to give the last 4 characters of the NRIC (i.e. partial NRIC) for verification purposes. This reduces the security risks if the data collected is unintentionally revealed.

Imagine that you work for a telephone company.

- When can you ask for someone's NRIC?
- What should the company do to ensure that personal data of customers are protected?

A handphone company can ask for your NRIC to verify your identify (e.g. to check that you are indeed the person registering for a new mobile phone). The company may obtain the NRIC number to run necessary checks. For example, by law, each person is allowed to register up to 3 prepaid cards. The company uses your NRIC number to check that you have not exceeded the limit.

The company should ensure that the data is stored securely, for example, encrypted and stored in an intranet rather than on the Internet. There should be user authentication required before someone is allowed to access the data.

[2] Note that PDPA does not apply to public agencies and organisations acting on behalf of them, thus, for example, the police can collect your personal information, including NRIC/FIN. Data collected by public agencies are protected by other acts.

Example: Do Not Call Registry

Have you ever received calls from unknown companies who seem to know your name and perhaps try to sell products to you? Your telephone number could have been gathered from unexpected sources, such as a lucky draw form that you filled up long ago. With technology, companies can easily gather and consolidate personal information. It can even automate the making of such calls.

To prevent you from getting unnecessary marketing calls, you can register in the **Do Not Call (DNC) Registry** to opt out of marketing messages and calls. The PDPA prohibits organisations from sending marketing messages to Singapore telephone numbers, including mobile, fixed-line, residential and business numbers that are registered with the DNC Registry.

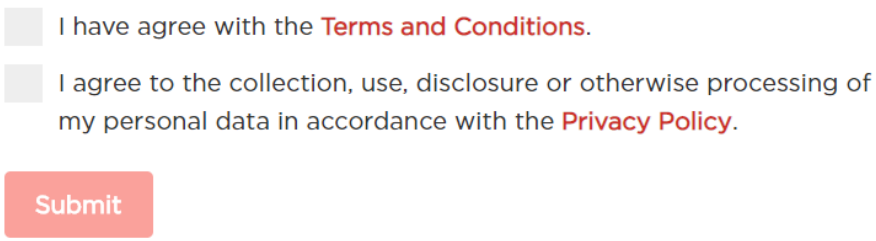
There are three DNC registers that individuals can choose to register in:

- No Voice Call Register
- No Text Message Register
- No Fax Message Register

Registering the phone number in each register is to opt out of receiving marketing messages through voice calls, text messages and fax messages.

Note that organisations that have an ongoing relationship with a subscriber or user of a Singapore telephone number may send marketing messages on similar or related products, services and memberships to that Singapore telephone number via text or fax without checking against the DNC Registry. However, each exempt message must also contain an opt-out facility that the recipient may use to opt out from receiving such telemarketing messages. If a recipient opts out, organisations must stop sending such messages to his/her Singapore telephone number after 30 days.

You can take various measures to protect your personal data. Do not reveal your personal data to unknown sources. For phone calls, ensure that the caller is who he or she is before giving your personal information. For websites and applications, read the privacy or data protection policies of the website to understand how your data are used before agreeing with the terms.



If you have queries on personal data, or to withdraw consent, you can contact the data protection officer (DPO). Under PDPA, companies are required to appoint one or more persons to be DPO to oversee the data protection responsibilities within the organisation and ensure compliance with the PDPA.

Also be careful when throwing away papers containing your personal data, such as application forms or letters from schools, banks etc. Tear or shred them so that people cannot use them to obtain personal data about yourself.

Let's Apply!
Read the following excerpt, which is an adaptation of an actual case, and answer the questions that follow.

- - -

A pre-school organised a school trip for interested students and their parents. To verify that only authorised parents turned up for the school trip, the pre-school teacher collected the parent's personal data (like identity card numbers).

A few days before the school trip, the teacher sent a file of the consolidated name list to the parents' WhatsApp chat group to remind those who signed up about the school trip. The file contained a table that included the names of the students, along with the contact number and identity card numbers of the parents attending.

- - -

(a) In what way was the PDPA breached in the scenario above?

The teacher released the personal data of some parents to other parents without their permission.

(b) What precautions can a teacher take to prevent a similar accident from happening?

The teacher should have messaged each parent directly to remind the parents of the school trip rather than messaging everyone in a group chat to minimise the possibility of accidental leakage of personal data.

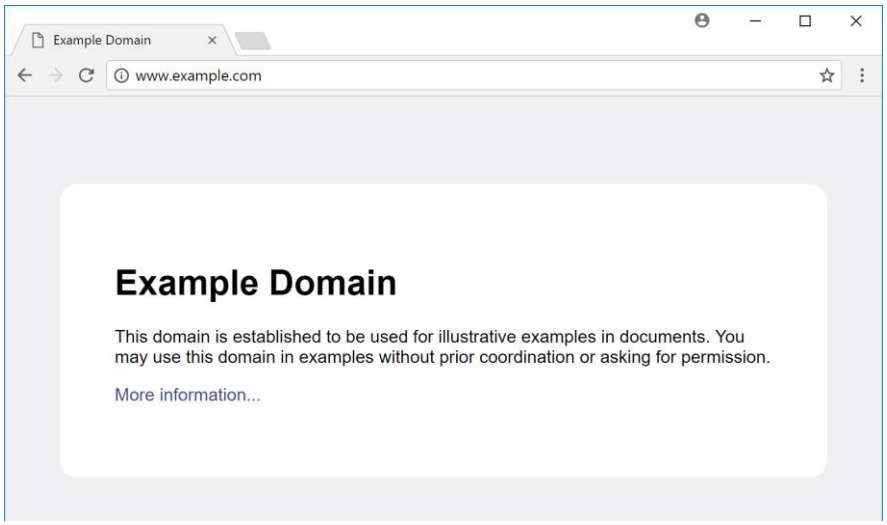
(c) What should the teacher do to the personal data obtained after the school trip?

The teacher should delete the personal data if they are not required after the school trip. Any printed copy of the data should be shredded to prevent leakage of personal data.

2021 JC2 H2 Computing 9569
27. Hypertext Markup Language (HTML)

Introduction

Open a web browser, such as Google Chrome, and visit www.example.com. You should see a simple web page as shown in the following screenshot.



How do you think web pages like this one are made?

Just like how programs are written in a programming language, such as Python, web pages are written using the **Hypertext Markup Language (HTML)**. Unlike programming languages that are specialised for describing step-by-step instructions, HTML is used to describe the structure of web pages. It provides control over how contents (e.g. words, images, sounds, etc.) are displayed on the website.

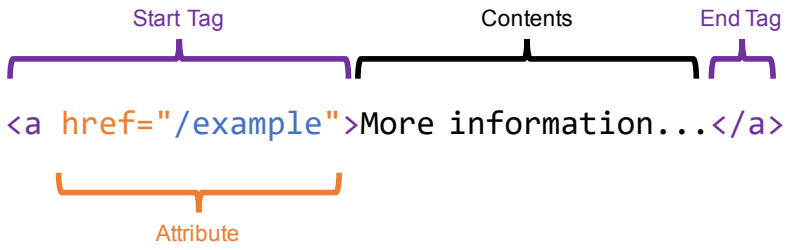
To view the HTML source code of a web page in Chrome, we can press Ctrl-U. Alternatively, we can do a right-click and select “View page source”. Try this now for the web page on www.example.com.

Examine the HTML source code on the web browser. Do you see that the contents of the web page are surrounded by text enclosed in angle brackets (i.e. < and >)? The text surrounded by angle brackets are special processing instructions for the web browser called **tags**.

You may also notice that there is a portion in the HTML document that does not include tags. That is written in the **Cascading Style Sheets (CSS)** language that controls the appearance of the web page. We shall cover CSS in the next chapter.

Anatomy of an HTML Document

Most tags consist of a **start tag** and an **end tag**. This is unlike Python that uses indentation to represent the start and the end of a block of code. Start tags may also have one or more attributes.



Example 1

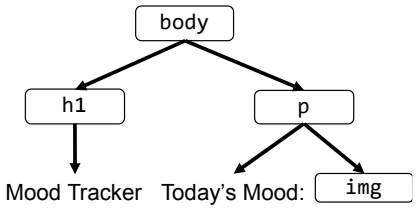
```
<!DOCTYPE html>

<html>
    <head>
        <title>Tab title</title>
    </head>
    <body>
        <h1>This is my 1st header.</h1>
    </body>
</html>

<!-- Add your comments here. -->
```

Tag	Purpose
`<!DOCTYPE>`	Declares the type and the version of the document. `<!DOCTYPE html>` indicates that it is written in HTML5
`<html>`	Contains the entire document
`<head>`	Contains the metadata of a document
`<body>`	Contains the visible elements of a document
`<title>`	Adds a title to the web page
`<h1>, <h2>, <h3>, <h4>, <h5>, <h6>`	Six levels of headings
`<!-- comments -->`	Comments

A start tag that comes with a matching end tag, such as `<body>` and `<h1>`, corresponds to a **normal element** that may contain a combination of text and others. On the other hand, a start tag that does not have a matching end tag, such as `<!DOCTYPE>` and `<img>`, corresponds to a **void element** that does not have any contents.

HTML tags are used to describe the structure of a web page by organising its contents into a tree of elements. In general, each start tag corresponds to a single element. Take a look at how the following HTML snippet is represented as a tree.

```
<body>
    <h1>Mood Tracker</h1>
    <p>
        Today's Mood:
        <img src="happy.jpg" alt="happy">
    </p>
</body>
```



Header Tags

Example 2

```
<!DOCTYPE html>

<html>
    <body>
        <h1>This is my 1st header.</h1>
        <h2>This is my 2nd header.</h2>
        <h3>This is my 3rd header.</h3>
        <h4>This is my 4th header.</h4>
        <h5>This is my 5th header.</h5>
        <h6>This is my 6th header.</h6>
    </body>
</html>
```

Notice how the headers are arranged in descending order of size from `h1` to `h6`.

Text Formatting Tags

Example 4

```
<!DOCTYPE html>

<html>
    <body>
        <h1>This is my 1st header.</h1>
        <p><u>This is my 1st paragraph.</u></p>
        <p>This is my 1st statement <br> and 2nd statement.</p>
        <hr>
        <p>This is my <b><i>last statement.</i></b></p>
    </body>
</html>
```

Tag	Purpose
<p>	Adds a new paragraph
<b>	Bolds texts
<i>	Italicises texts
<u>	Underlines texts
 	Adds a line break
<hr>	Adds a horizontal line and line break

Note that some characters have special meanings in HTML. To display them as texts, they need to be escaped using character references that start with an ampersand (&) and end with a semi-colon (;).

Character	&	<	>	"
Character reference	&amp;	&lt;	&gt;	&quot;

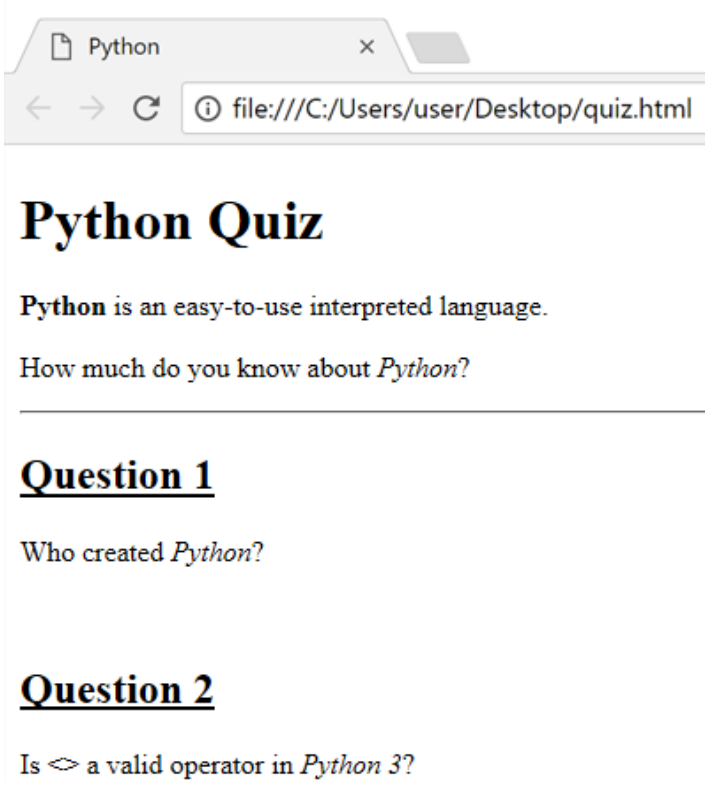
Some browsers may be able to interpret the above characters correctly and display them as intended without using the character references. However, it is best practice to use the character references to avoid ambiguity.

Quick Check

1. Underline the tags having normal elements and circle the tags having void elements.

```
<!DOCTYPE html>

<html>

<head><title>Welcome Page</title></head>

<body>

            <h1>Welcome to our Computing department!</h1>

            <img src="welcome.png" alt="Welcome">

            <p>Feedback:</p>

            <textarea name="feedback">Type here.</textarea>

            <p><input type="submit"></p>

</body>

</html>
```

2. Create the following web page. Save your file as `quiz.html`.



Unordered and Ordered Lists

Can you spot the difference in the next two examples?

Example 5

```
<!DOCTYPE html>

<html>
    <body>
        <h1><u>About me</u></h1>
        <p>I am a student studying in ACJC.</p>
        <p>The following are my hobbies:</p>
        <ul>
            <li>Eating</li>
            <li>Sleeping</li>
            <li>Watching TV</li>
        </ul>
    </body>
</html>
```

Example 6

```
<!DOCTYPE html>

<html>
    <body>
        <h1><u>About me</u></h1>
        <p>I am a student studying in ACJC.</p>
        <p>The following are my hobbies:</p>
        <ol>
            <li>Eating</li>
            <li>Sleeping</li>
            <li>Watching TV</li>
        </ol>
    </body>
</html>
```

Tag	Purpose
`<ul>`	Creates an unordered list where items are marked with bullet points
`<ol>`	Creates an ordered list where items are marked with numbers
`<li>`	Marks the individual items in a list

Tables

Example 7

```
<!DOCTYPE html>

<html>
    <body>
        <h1>About me</h1>
        <p>I am a student studying in ACJC.</p>

        <h2>Here are my O-Level results:</h2>
        <table>
            <th>Subject</th>
            <th>Grade</th>
            <tr>
                <td>English Language</td>
                <td>A1</td>
            </tr>
            <tr>
                <td>Mathematics</td>
                <td>A1</td>
            </tr>
        </table>
    </body>
</html>
```

Tag	Purpose
`<table>`	Creates a table
`<th>`	Table header
`<tr>`	Table row
`<td>`	Table data within a row

Note that CSS is required to show table borders.

Images and Links

Search for a picture of a dog from the Internet and save it as `dog1.jpeg`. Afterwards, create the following HTML document and put it in the same directory as the image file.

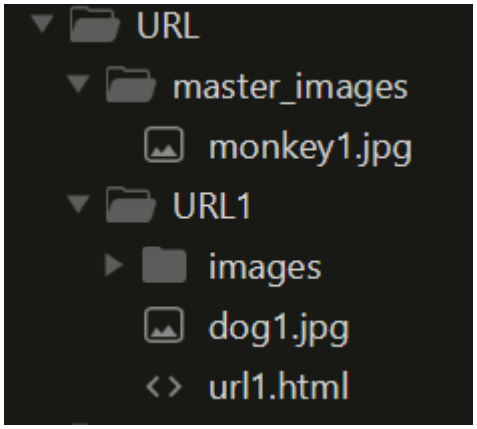
Example 8

```
<!DOCTYPE html>

<html>
    <body>
        <img src="dog1.jpeg" alt='Image of a dog'>
        <br>
        <img src="cat1.jpeg" alt='Image of a cat'>
        <br>
        <a href="http://www.google.com">Go to google.com</a>
    </body>
</html>
```

Tag	Purpose
`<img>`	Displays an image `src=` is followed by the (path of the) filename in quotation marks `alt=` is followed by a text to be displayed if there is an issue with the file
`<a>`	Anchor tag to create a link `href=` is followed by the URL in quotation marks When creating a link to an external website, the link provided must be an absolute URL. It is necessary to include the “http://” to ensure that the browser does not interpret it as something that resides in our file system.

Example 9

Create a directory with the following structure and include the necessary files. The following HTML code is to be included in a HTML document named `url1.html`.




```
<!DOCTYPE html>

<html>
    <head>
        <title>URL1</title>
    </head>
    <body>
        <img src="dog1.jpeg" alt='Image of a dog'>
        <br>
        <img src='images/cat1.jpeg' alt='Image of a cat'>
        <br>
        <img src='../master_images/monkey.jpeg' alt=''>
    </body>
</html>
```

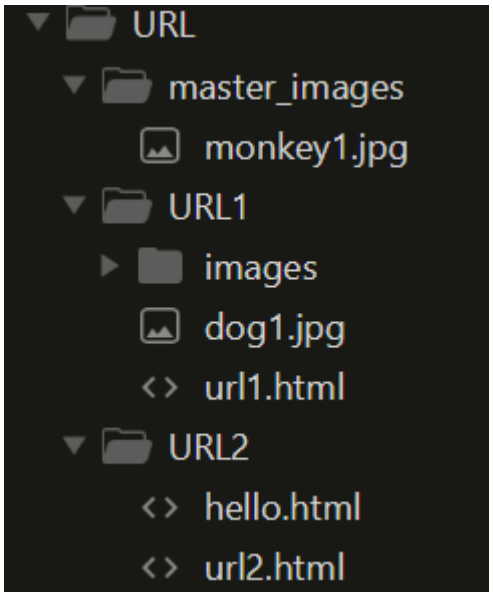
If the path is not specified, it is assumed that the image file is in the same folder as the HTML document.

If the image file is located in a sub-folder inside the same folder as the HTML file, the path should be specified in the following format: "`folder_name/image_name`"

If the image is located in a folder that is outside of the one where the HTML file resides in, we can use the "`..`" to go to the parent directory, i.e. one level up.

Example 10

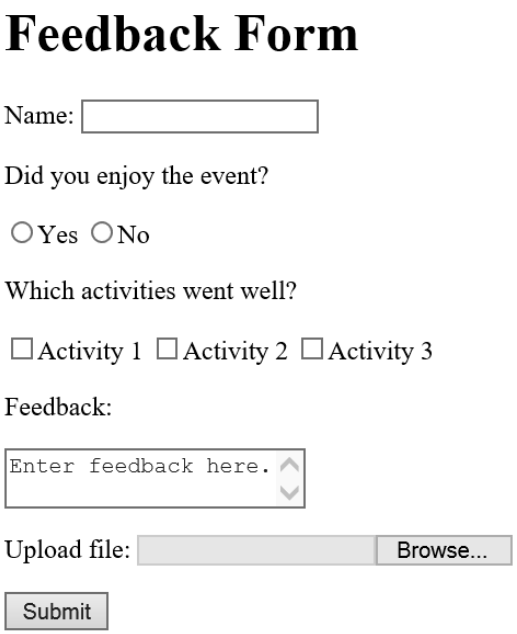
To the directory in Example 9, let us now include a folder named `URL2`. Create a new HTML document to your liking and name it `hello.html`. The following HTML code is to be included in a HTML document named `url2.html`.




```
<!DOCTYPE html>

<html>
    <head>
        <title>URL2</title>
    </head>
    <body>
        <a href="http://www.google.com">This is a link to Google</a>
        <br>
        <a href="hello.html">This links to hello.html</a>
        <br>
        <a href="../URL1/url1.html">This links to url1.html</a>
    </body>
</html>
```

Forms

A **form** on a web page allows users to enter data that are sent to a server for processing. An example of a simple form is shown below.



Example 11

```
<!DOCTYPE html>

<html>
    <head>
        <title>Form 1</title>
    </head>
    <body>
        <form action='http://www.example.com'>
            <h1>Feedback Form</h1>

            <p>Name: <input name="username" type="text" value=""></p>

            <p>Did you enjoy the event?</p>
            <input type="radio" name="choice1" value="yes">Yes
            <input type="radio" name="choice1" value="no">No<br>

            <p>Which activities went well?</p>
            <input type="checkbox" name="choice2" value="act1">1st
            <input type="checkbox" name="choice2" value="act2">2nd
            <input type="checkbox" name="choice2" value="act3">3rd<br>

            <p>Feedback:</p>
            <textarea name="feedback">Enter feedback here.</textarea>

            <p>Upload file: <input name="some_file" type="file"><br>

            <p><input type="submit" value="Submit"></p>
        </form>
    </body>
</html>
```

Each form is contained in a separate `<form>` tag with an `action` attribute set to the URL where the submitted data will be sent to. In future practical tasks, we will learn how to write a Python program that runs on a web server to process the submitted data.

Inside the `<form>` tag, each `<input>` and `<textarea>` tag represents an **input control** with a unique `name` attribute to allow the server to retrieve these inputs.

Tag	**Purpose**
`<input type = 'text'>`	Creates a text field
`<input type = 'checkbox'>`	Creates a checkbox
`<input type = 'radio'>`	Creates a radio button
`<input type = 'file'>`	Creates a field for the uploading of file
`<input type = 'submit'>`	Creates a submit button
`<input type = 'hidden'>`	Creates a hidden text field, which is typically used to include information not to be seen by the user, e.g. which database to update the submitted information into
`<textarea>`	Creates a multi-line text box `rows=x` and `cols=y` can be used to specify the number of rows and columns respectively

Grouping of HTML Code

The example below shows how `<div>` can be used to organise HTML code into blocks, which is particularly useful when we want to add a style to one part of our web page.

Example 12

```
<!DOCTYPE html>

<html>
    <body>
        <div style="color:red">
            <h1>This is my 1st division.</h1>
            <p>I have some words here.</p>
        </div>
        <div style="color:green">
            <h1>This is my 2nd division.</h1>
            <p>I have more words here.</p>
        </div>
    </body>
</html>
```

Example 13

Another tag, `<span>`, achieves a similar outcome, but it is typically used to separate a line of code into multiple parts.

```
<!DOCTYPE html>

<html>
    <body>
        <p>Span is used to break up <span style="color:red"> a long
           line    of    code</span>    into    multiple    portions,
           <span    style="color:green">    each    with    their    own
           styling.</span></p>
    </body>
</html>
```

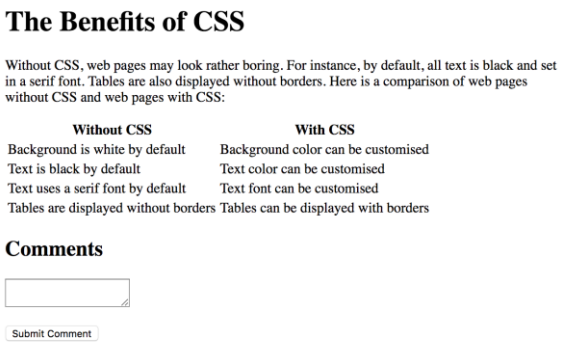
References

HTML tags: https://www.w3schools.com/tags/default.asp
HTML special characters: https://www.w3schools.com/html/html_entities.asp

2021 JC2 H2 Computing 9569
28. Cascading Style Sheets (CSS)

Introduction

We have been able to create simple web pages using HTML such as the one shown below.



The Benefits of CSS

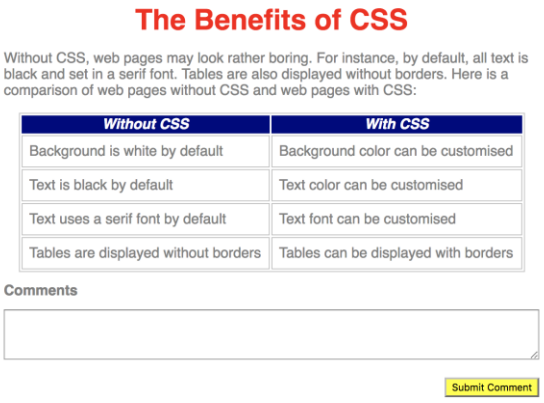
Without CSS, web pages may look rather boring. For instance, by default, all text is black and set in a serif font. Tables are also displayed without borders. Here is a comparison of web pages without CSS and web pages with CSS:

Without CSS	With CSS
Background is white by default	Background color can be customised
Text is black by default	Text color can be customised
Text uses a serif font by default	Text font can be customised
Tables are displayed without borders	Tables can be displayed with borders

Comments

Submit Comment

While it is possible to add styles directly into our HTML code, it makes the code long and difficult to read. With the help of Cascading Style Sheets (CSS), we can improve the appearance of web pages greatly with almost no change to the HTML code.



The Benefits of CSS

Without CSS, web pages may look rather boring. For instance, by default, all text is black and set in a serif font. Tables are also displayed without borders. Here is a comparison of web pages without CSS and web pages with CSS:

Without CSS	With CSS
Background is white by default	Background color can be customised
Text is black by default	Text color can be customised
Text uses a serif font by default	Text font can be customised
Tables are displayed without borders	Tables can be displayed with borders

Comments

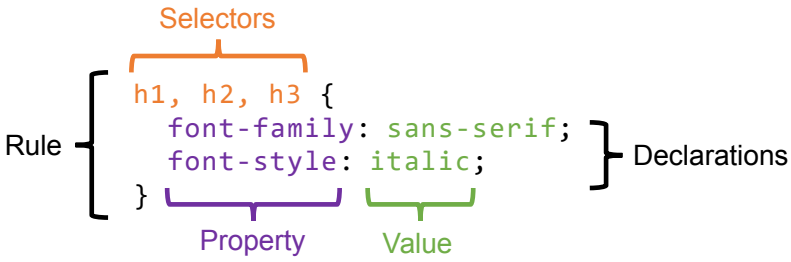
Submit Comment

Just like how abstraction is useful when we do Python programming, a computer science principle called **separation of concerns** applies here, where a program is divided into distinct sections such that each section only deals with one aspect of the final product and has minimal knowledge of the other parts.

Take a look at www.csszengarden.com. Click on any of the available designs to see how changing CSS can dramatically affect the appearance of a web page without modifying its HTML. You may view the source code to check that the HTML code is exactly the same for each design.

Anatomy of CSS

CSS is made up of multiple **rules**. Each rule starts with one or more **selectors** separated by commas, followed by curly braces surrounding a number of **declarations**. Each declaration is made of two parts: a **property** name and one or more values separated by spaces. Multiple declarations in a rule are separated by semicolons.



Example 1

Use Notepad++ to type the CSS code below and save it in a folder as `style1.css`.

```
h1 {
    background: red;
    color: blue;
    text-align: right;
}

h2, h3 {
    font-family: sans-serif;
    font-style: italic;
    font-size: 36px;
}
```

We also need a HTML code in the same folder that uses the CSS file above.

```
<!DOCTYPE html>

<html>
    <head>
        <title>Example 1</title>
        <link rel='stylesheet' href='style1.css'>
    </head>
    <body>
        <h1>This is my 1st header.</h1>
        <h2>This is my 2nd header.</h2>
        <h3>This is my 3rd header.</h3>
    </body>
</html>
```

Notice that we have added a `<link>` tag under the head element. The `rel` attribute (short for 'relationship') has a value of `'stylesheet'`, which indicates that the hyperlink a CSS file.

Colours

As seen in the Example 1, `background` and `color` properties to set the background and text colour of elements respectively.

The following are some of the colour names that can be used for the two properties

red	orange	yellow	green	blue	purple
black	gray	silver	white	transparent	
				(no colour)	

If a desired color does not match any of the above names, we can also specify a colour in terms of its **RGB** (red, green and blue) components. Each component is expressed as an integer between 0 to 255 (inclusive) and the color is written as `rgb(R, G, B)`. For example, the following CSS code sets the page background to a shade of pale yellow.

```
body { background: rgb(255, 255, 128); }
```

The same color can be expressed as three hexadecimal numbers of two digits each (including a leading zero if needed). The color can thus be written as `#RRGGBB`, where `RR` is the red component, `GG` is the green component and `BB` is the blue component, all in hexadecimal. For example, the same shade of pale yellow can also be written as follows.

```
body { background: #ffff80; }
```

Notice that `ff` is the integer 255 in hexadecimal, while `80` is the integer 128 in hexadecimal. Also note that the hexadecimal digits are not case sensitive.

For convenience, if each of the three hexadecimal numbers is made of repeated digits (e.g. `00`, `11`, `22`, ..., `FF`), then the colour can be shortened to `#RGB`. For example, while `#FFFF80` cannot be shortened, the color `#00FFCC` can be shortened to `#0FC`.

Typography

CSS	Result
`p { font-family: serif; }`	This is an example of a paragraph.
`p { font-family: sans-serif; }`	This is an example of a paragraph.

The `font-family` property specifies which typeface is used to display the text. A **serif** font such as 'Times New Roman' has lines extending from the ends of each letter stroke. Such fonts are traditionally used for long pieces of printed text. A **sans-serif** font such as 'Arial', however, does not have these additional lines. The browser will use the first font in the list that is installed. A specific font name can also be specified. In such a case, it must be enclosed in quotation marks, e.g. 'Comic Sans MS'.

CSS	Result
`p { font-size: 24px; }`	This is an example of a paragraph.

The `font-size` property can be used to specify text size in pixels.

CSS	Result
`p { font-style: italic; }`	*This is an example of a paragraph.*
`p { font-weight: bold; }`	**This is an example of a paragraph.**
`p { font-style: italic; font-weight: bold; }`	***This is an example of a paragraph.***

The `font-style` property specifies whether an italic font is used (i.e. `normal` or `italic`). On the other hand, the `font-weight` property specifies whether a bold font is used (i.e. `normal` or `bold`).

CSS	Result
`p { text-align: left; }`	This is an example of a paragraph with enough content to see how text-align works.
`p { text-align: center; }`	This is an example of a paragraph with enough content to see how text-align works.
`p { text-align: right; }`	This is an example of a paragraph with enough content to see how text-align works.
`p { text-align: justify; }`	This is an example of a paragraph with enough content to see how text-align works.

The `text-align` property specifies how the text is aligned.

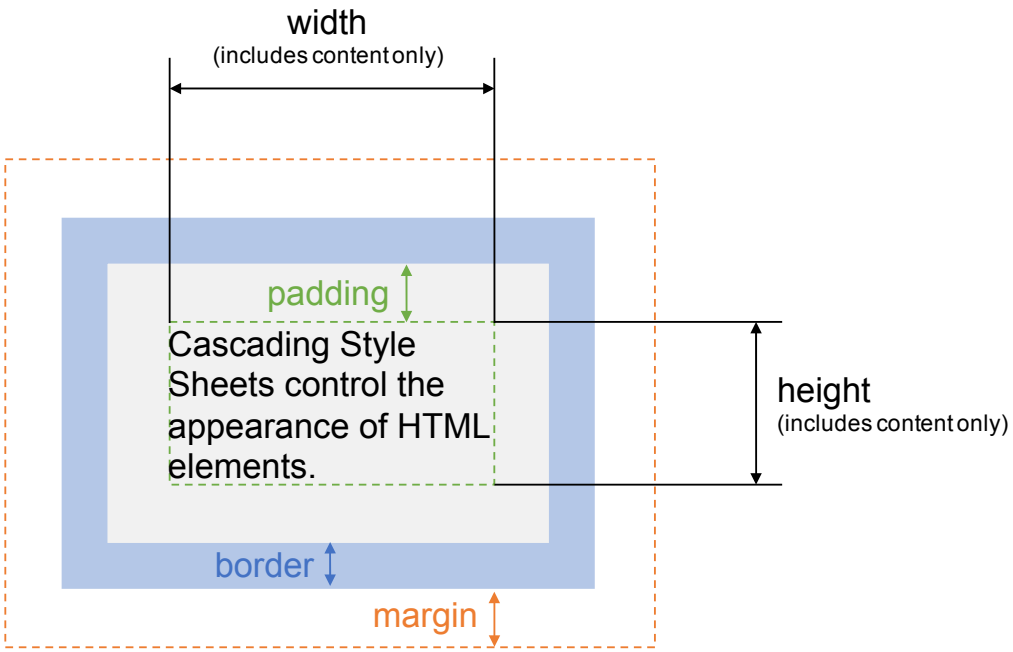
CSS	Result
`p { text-decoration: underline; }`	This is an example of a paragraph.
`p { text-decoration: line-through; }`	~~This is an example of a paragraph.~~

The `text-decoration` property specifies whether additional elements of the font are displayed. The most common values of this property are `none`, `underline` and `line-through`.

Box Model

Notice that some HTML tags such as `<h1>` and `<p>` always start on a new line and force the following element to also start on a new line. On the other hand, tags such as `<b>` and `<i>`, do not. This is because the former have a **block appearance** by default, while the latter have an **inline appearance** by default.

The **box model** is illustrated below.



Property Name	Description
`border`	Specifies the thickness of the optionally-coloured border around the element
`margin`	Specifies the thickness of the transparent space surrounding the border
`padding`	Specifies the thickness of the space between the content and the border that is filled with the element's background colour or pattern
`width`	Specifies the element content's width, regardless of the surrounding margin, border and padding
`height`	Specifies the element content's height, regardless of the surrounding margin, border and padding

When setting a box's width and height or the thickness of its margin, border and padding, we must specify a unit of measurement, commonly in pixels.

CSS	Result
`p { border: 5px solid red; }`	This is an example of a paragraph with a 5 pixel thick red border written using the <p> tag.

To specify that a border should be drawn with a solid colour, we use the value of a thickness, followed by a space, the word `solid`, another space and finally the colour we wish to use for the border.

CSS	Result
`p { border-bottom: 1px solid gray; }`	This is an example of a paragraph.
`p { border-left: 1px solid gray; }`	This is an example of a paragraph.
`p { border-top: 1px solid gray; }`	This is an example of a paragraph.
`p { border-right: 1px solid gray; }`	This is an example of a paragraph.

By default, the margin, border and padding properties control the appearance for all four sides of the element's box. However, we can append `-bottom`, `-left`, `-top` or `-right` to any of these properties so that we control the appearance for only one side of the box.

CSS	Result
`p { background: silver; margin-left: auto; margin-right: auto; width: 100px; }`	This is an example of a paragraph that is forced to be only 100 pixels wide and centered.

Example 2

Use Notepad++ to type the CSS code below and save it in a folder as `styles2.css`.

```
table {
    border: 1px solid black;
    width: 50%;
    height: 30px;
    border-collapse: collapse;        <!-- What is this used for? -->
}

th,td {
    border: 1px solid red;
    text-align: center;
}
```

By using the CSS code above, we can create proper tables defined by the following HTML code.

```
<!DOCTYPE html>

<html>
    <head>
        <title>Example 2</title>
        <link rel='stylesheet' href='style2.css'>
    </head>
    <body>
        <h2>O-Level Results</h2>
        <table>
            <th>Subject</th>
            <th>Grade</th>
            <tr>
                <td>English Language</td>
                <td>A2</td>
            </tr>
            <tr>
                <td>Mathematics</td>
                <td>A1</td>
            </tr>
            <tr>
                <td>Computing</td>
                <td>A1</td>
            </tr>
        </table>
    </body>
</html>
```

Types of Selectors

The elements that are affected by each CSS rule are determined by the selectors at the start of that rule. We shall look at four types of selectors.

Element Selector

An **element selector** picks out all elements of a particular type from the HTML document. We have been using this particular selector in the examples thus far. An example is shown below.

```
p { color: blue; font-style: italic; }
```

Based on this rule, all `p` elements on the web page will appear as blue italic text.

What if we only want selected parts to be stylised? There is a way in which we can fine-tune our selection by making use of two special attributes that are valid for all HTML tags.

Id Selector

An **id selector** picks out the unique element that has a particular value for its `id` attribute. To use an id selector, we enter a hex symbol (#) followed immediately by the desired element's `id` attribute value. Since `id` attributes on a web page cannot be repeated, an id selector will always pick out exactly one element if it exists.

For instance, suppose we have the following HTML and CSS files in the same folder.

id-example.html

```
<!DOCTYPE html>

<html>
  <head>
    <title>ID Selectors Example</title>
    <link rel="stylesheet" href="id-example.css">
  </head>
  <body>
    <p>This is a normal paragraph.</p>
    <p id="special">This paragraph is special.</p>
    <p>This is a normal paragraph.</p>
  </body>
</html>
```

id-example.css

```
#special { color: red; }
```

If we open `id-example.html`, we see that only the second `p` element with an `id` of `"special"` is formatted as red.

Class Selector

A **class selector** picks out all elements that are associated with a particular class. To use a class selector, we enter a period (.) followed immediately by the class name we wish to reference.

For instance, suppose we have the following HTML and CSS files in the same folder.

class-example.html

```
<!DOCTYPE html>

<html>
  <head>
    <title>Class Selectors Example</title>
    <link rel="stylesheet" href="class-example.css">
  </head>
  <body>
    <p>This is the main paragraph.</p>
    <p class="info">This is the first information.</p>
    <p class="info">This is the second information.</p>
  </body>
</html>
```

class-example.css

```
.info { color: silver; }
```

If we open `class-example.html`, we see that only the second and third `p` elements with a `class` of `info` are formatted as silver.

Descendent Selectors

Sometimes, it is necessary to select an element only if it has a parent element that matches another selector. This can be achieved using the **descendent selector**. To use a descendent selector, separate any two selectors using a space: the corresponding rule will only be applied for elements that match the selector on the right *and* have a parent element that matches the selector on the left.

For instance, suppose we have the following HTML and CSS files in the same folder.

descendent-example.html

```
<!DOCTYPE html>

<html>
  <head>
    <title>Descendent Selectors Example</title>
    <link rel="stylesheet" href="descendent-example.css">
  </head>
  <body>
    <h1>Heading with <i>Italics</i></h1>
    <p>This paragraph has <i>italics</i>.</p>
    <i>Bare italics</i>
</body>
</html>
```

descendent-example.css

```
p i { color: red; }
```

If we open `descendent-example.html`, we should see the following.

Only the `p` element has its italics portion formatted as red. The other `i` elements remain unformatted as they do not match the specific requirements of the selector, which requires the `i` element to be a descendent of a `p` element.

References

CSS properties: https://www.w3schools.com/cssref/default.asp
CSS selector: https://www.w3schools.com/cssref/css_selectors.asp

29 Computer Networks

Introduction

A computer does not need to be a stand-alone device. It can be connected with other computers. For example, computers in a classroom can be connected to each other to share files. When this is done, they form a **network**.

A network allows computers to

- communicate with one another
- share information centrally
- share copies of software
- give access to data and program files to multiple users.

On the other hand, because it is more difficult to control access to a network, the files stored on a network are also less secure than files stored on a standalone computer. Network security will be discussed in a future chapter.

Local Area Networks (LANs)

In a LAN, the computers are usually in the same building, or even the same room. In addition to sharing files, they can also share hardware such as printers, scanners, and other peripherals.

Typically, computers in the LAN are connected using cables or wireless signals. As electrical signals deteriorate as they travel along a cable, there is a maximum length for the cable (about 300m). For a wireless LAN, there needs to be a central router that broadcasts a signal to which all computers on the network connect. Security systems need to be put in place to ensure that unauthorised computers do not connect to the wireless network.

A device is also needed for the central storage of files. This is carried out by a computer that controls the network, which is known as a **server**. The server plays additional roles as well.

- A file server is responsible for storing program files, the network operating system, and users' data files
- A domain controller server is responsible for the authentication of user log-ons.
- A print server is responsible for managing shared devices.

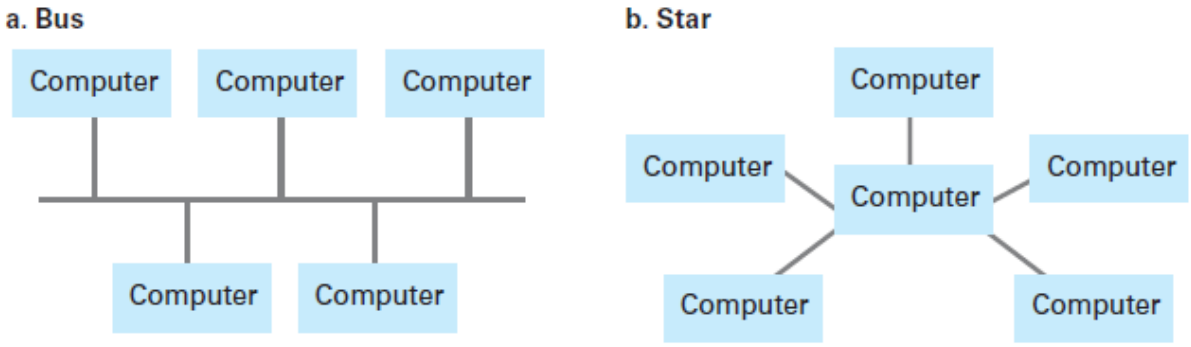
In a small network with a few computers, these functions may be carried out by a single network server. If the server fails, then all work has to be stopped across the LAN. This makes the server a vulnerability. The other computers in the network are known as **clients**.

The communications around such a system are difficult to control, therefore all the computers in the network must follow a set of instructions. The **network operating system** provides those instructions, carrying out tasks such as

- controlling access to the network
- management of the filing system
- management of all applications and programs available from the server
- management of all shared peripherals e.g. printers.

Structure and hardware in a LAN

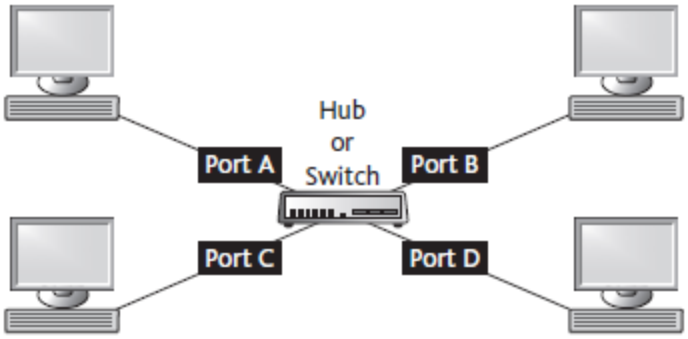
The way that computers are connected in a network is called its **topology**. Each topology has its own advantages and disadvantages, and this affects both the hardware components that would be used. Two common topologies are the **bus** and **star** topologies. Note that the pictures illustrate the connections between the computers, and may not reflect how the computers are physically placed relative to each other.



In a bus topology, all the computers are connected to a central communication line, called the **bus**. The bus topology was popularised in the 1990s when **Ethernet** arose as a standard for communication.

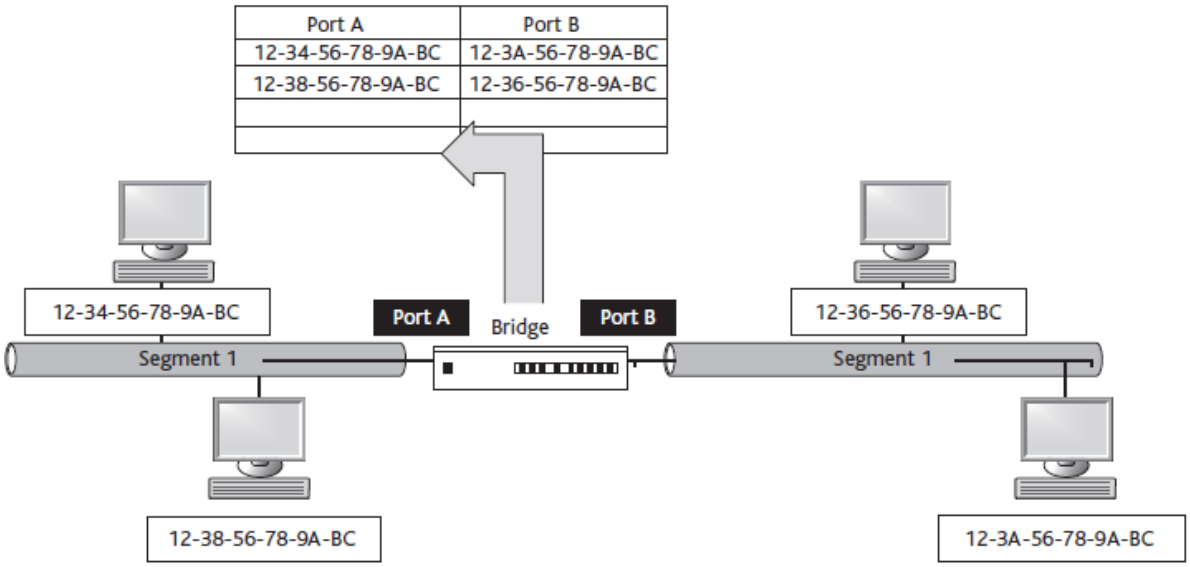
Star topologies have been used since the 1970s when the common paradigm was that of a large central computer serving many users. As each users' individual terminal grew into a full computer, a star topology emerged naturally. Nowadays, it is common in wireless networks where communication is by means of radio broadcast and the central machine, or **access point**, is the focus around which all communication is coordinated.

Bus and star networks may be connected into bigger, more complicated networks.

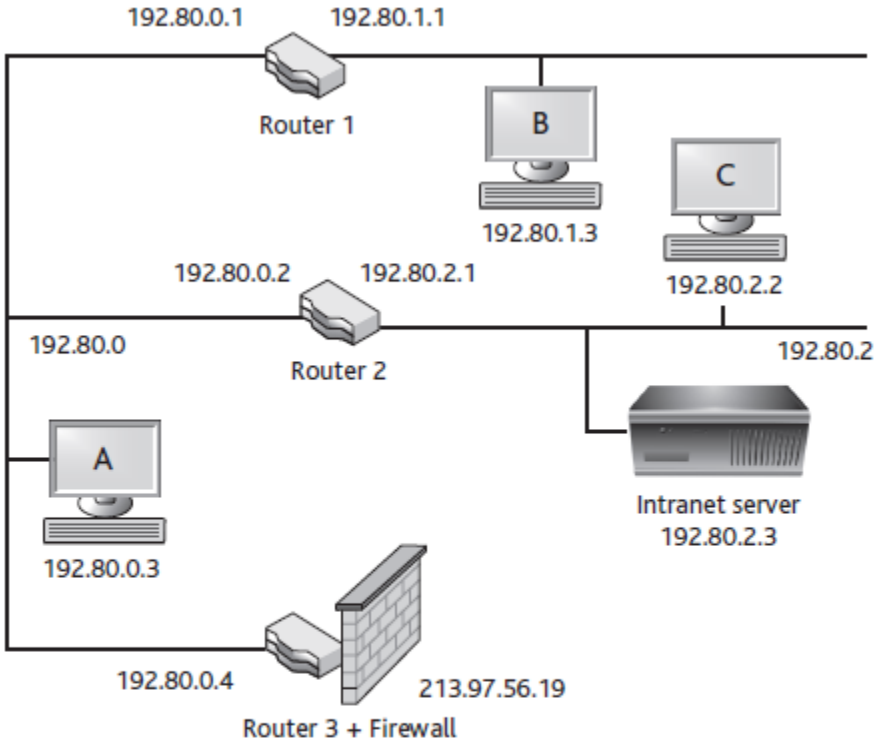


A **hub** or receives all the data from individual computers and broadcasts them back to all the devices on it.

A **switch** is more sophisticated, as it reads the destination label of the data and sends it only to the device for which the data is intended. This reduces the amount of traffic on the network. It sets up a temporary dedicated circuit between the sender and receiver, and releases the circuit once the data is transferred.



A **bridge** connects two LAN segments. Each device has a **MAC (media access control) address** (an address or serial number given to it by the manufacturer) and the bridge maintains a table showing which MAC addresses are connected to which port. It does not vet the data's content to see whether it should be transferred.



Host Name	IP address	MAC address
Web Server	213.97.56.20	00-BF-99-20-56-13
Router 1	192.80.0.1	00-09-94-39-3B-9A
Router 2	192.80.0.2	21-00-98-8D-66-E2

A **router** is similar to a bridge, but it also exercises a degree of decision making. It can decide, based on the sender and the receiver, whether to allow data to be transmitted from one device to another. It can therefore be used as a security device.

Finally, a **gateway** connects a LAN to a WAN (possibly the Internet). It ensures that data transmitted between one side and the other is appropriate and monitors the usage of the connection. It can be considered a single point of entry to a LAN from a larger network.

Wide Area Networks (WANs)

In a WAN, the distances between the computers are much further. A WAN may be spread across a country or even internationally. It is thus not possible to connect the computers directly using cables or wireless signals.

One way that computers can be connected is by using existing infrastructure such as the telephone network.

In the past, the **digital** electrical signals produced by a computer were different from the **analog** signals transmitted by the telephone lines are different. A device called a **modem** (short for **mo**dulator-**dem**odulator) was needed to convert the digital computer signals to analog signals for the telephone network. At the receiving end, another modem would convert the analog signals back to digital signals for the receiving computer.

The table below shows a comparison between a typical LAN and a typical WAN today.

LAN	WAN
It is used by an organisation or company within a site or branch	It is used by an organisation or company to connect sites or branches
It is owned by the organisation or company	It is be owned by the organisation or company
It is one of many individual LANs at one site	It is leased from a public switched telephone network (PSTN) company
	A dedicated communication link is provided by the PSTN
The transmission medium is twisted pair cable or WiFi	The transmission medium is fibre-optic cable
The LAN contains a device that allows connection to other networks	Transmission within the WAN is from switch to switch
	A switch connects the WAN to each site
There are end-systems connected which are user systems or servers	There are no end-systems connected to the WAN

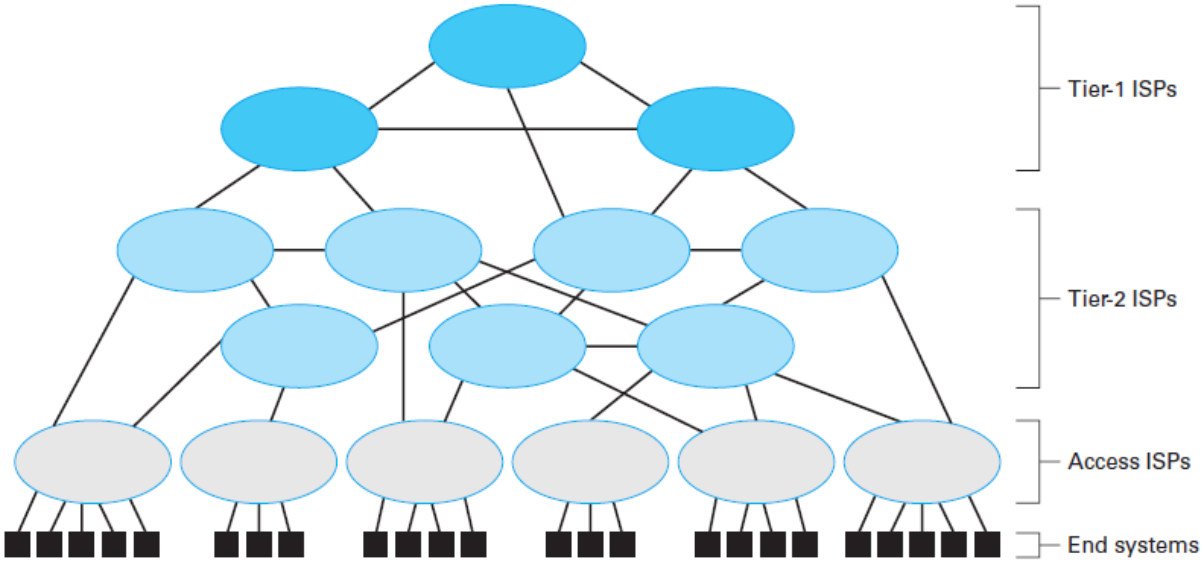
The Internet

One of the WANs developed in the US in the 1970s was known as ARPANET, named after the Advanced Research Projects Agency (ARPA) in the US Department of Defense. This network comprised mostly computers in military installations and research universities.

In the 1980s, the widespread use of Personal Computers (PCs) led to the creation of the first LANs. Over time, many of these networks, which were originally designed as independent, stand-alone networks, were eventually linked to each other, creating an inter-networking, which was shortened to **Internet**.

The Internet can be described as a WAN, but this severely understates its size and complexity. Furthermore, it is not centrally designed or organised, but evolved organically to arrive and its current form and will continue evolving in the future. Therefore, there is no agreed definition of its structure.

However, a hierarchy does exist within the structure of the Internet. An **Internet Service Provider (ISP)** was originally conceived to give Internet access to an individual or company. These are now called access ISPs, and they connect to middle tier or regional ISPs which are in turn connected to first tier or backbone ISPs. An ISP is a network and connections between ISPs are handled by **Internet Exchange Points (IXPs)**. The first tier ISPs and content providers can be considered to be at the top of the hierarchy.



While it is common in everyday language to talk about the Internet and the **World Wide Web (WWW)** as the same thing, this is not correct. The World Wide Web is a distributed application available on the internet. Specifically, it consists of a (very large) collection of websites, each of which contains one or more webpages.

The Internet has the following functions:

- Providing content from the World Wide Web
- Electronic mail
- File transfer

Intranets

An **intranet** is a network offering the same facilities as the Internet but solely from within a particular organisation. Information is made available from a web server and clients access material using web browser software.

Access to the intranet is usually restricted to people within the organization, and security can be ensured by using passwords and secure transmission lines. Different levels of password can be used to ensure that only specific people can access specific facilities on the intranet.

As there is a smaller volume of content on an intranet, it is more likely to be relevant to the organization. Furthermore, the amount of control means it is more likely to be correct, relevant and updated. As membership is restricted (and users could be identified), this means that comments are also more likely to be relevant and sensible

Some parts of an intranet may be made available to outside users. This access is called an **extranet**.

Communication protocols

When data is being transferred in a computer system, rules need to be set up for how the transfer is to be done. The set of rules is known as a **protocol**.

Some items covered by protocols include:

- The wire connecting two parts of the system, and the type of connections used
- The bit rate used (the rate at which data is being sent and received) must be the same for the sender and receiver
- The parity used (the system being used to check for mistakes in data transmission)

When two devices communicate with each other, the initial contact is a signal called a **handshake signal**. This is data exchanged so that both devices can establish that they are ready for the communication to start and they agree on the rules being used.

Informally, a data transmission between two devices may look something like this:

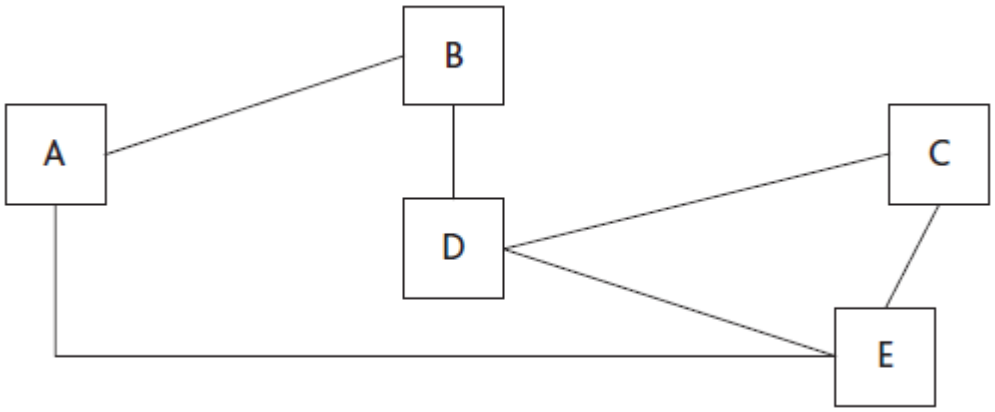
Sender:	"Hello Receiver, I have data for you."
Receiver:	"Hello Sender, I am ready."
Sender:	"Here comes the data."
Sender:	`DATA`
Sender:	"That was the data. Did you receive it?"
Receiver:	"I received something but I think it's wrong."
Sender:	"Here comes the data (again)."
Sender:	`DATA`
Sender:	"That was the data. Did you receive it?"
Receiver:	"Yes, it's correct. I'm ready for more data."
etc.	

The communication between the sender and the receiver which is not the actual `DATA` consists of handshaking and other overheads. In some networks, as much as 40% or more of the transmitted data consists of these overheads. This increases the time needed to move `DATA` through the network, but it is necessary to ensure that the message is received correctly.

Packet switching

When a message is sent from one computer to another (particularly over a WAN), the computers may not be directly connected to each other. The message would then have to pass through other devices.

For example, the diagram below shows a network. The devices, labelled A to E, are called **nodes**, and are connected via communication lines.



It is easy for A to send a message to B or E as A is connected to both B and E directly. However, there is no direct connection between A and C, therefore, other nodes are needed to **relay** the message. However, other nodes may also be sending messages to one another on the same network at the same time, and each communication line can only be used for one message at a time. To ensure that messages to not get confused, garbled or lost, it is necessary to have a system to relay messages so that they reach their destination.

Two common methods for doing this are packet switching and circuit switching.

In **packet switching**, the message is split into a number of equally-sized packets (also called **datagrams**). Each packet is given a label which consists of the address of the destination, and a packet sequence number. These packets are sent along the communication lines to the destination. Each time a packet reaches a node (an intersection point), the node decides which direction to send it on to.

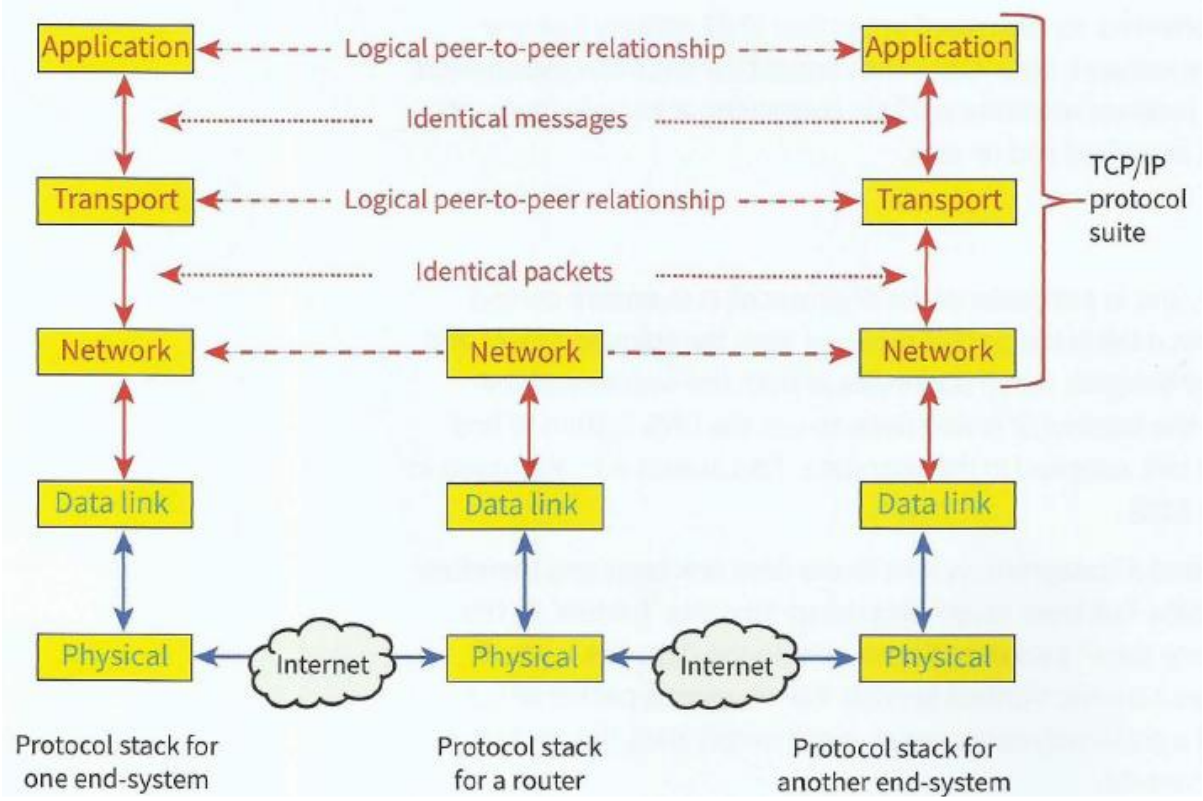
For example, one packet may go from A to E. If E can send it to C directly, E does so. However, the connection between them may be used for another message at the same time. In this case, E may choose to wait, or send it to D, which can send it to C.

When C receives all the packets, they are likely to be out of sequence. C needs to reassemble the original message in the correct order.

In **circuit switching** (out of syllabus), the network reserves a route from A to C. The message is then sent from A to C (via the relay nodes) as one continuous message and does not need to be reassembled when it arrives. However, circuit switching means that part of the network cannot be used by anyone else for the duration of the transmission.

TCP/IP protocol suite

A collection of related protocols is known as a **protocol suite**. The dominant protocol suite for Internet use is known as TCP/IP, which can be explained using the diagram of a network below.



The figure shows a stack of layers for a protocols where:

- Each layer except the physical layer represents software installed on an end-system or a router
- The software for each layer must provide the capability to receive and to transmit data in full-duplex mode to an adjacent layer. This means that data can be transmitted in both directions simultaneously.
- A protocol in an upper layer is serviced by protocols in the lower layers.

The roles of the layers are summarized as follows:

- The **Physical** layer protocols specify details about the transmission medium and hardware, e.g. electrical properties, radio frequencies, and signals.
- The **Data Link** layer consists of the network software that actually transfers data. It must deal with communication details particular to the individual network in which the computer resides. Specifications about network addresses, packet size, error reporting, protocols used to access the underlying medium, and hardware addressing are here.
- The **Network** layer protocols specify details about communication between two computers across multiple networks (e.g. across the Internet). The Internet addressing structure, the format of the Internet packets, the method of dividing a large Internet packet into smaller packets for transmission, and mechanisms for reporting errors are here. Examples of Network later protocols are IP, IGMP, ICMP, ARP.
- The **Transport** layer protocols provide for communication from an application on one end-system to an application on another end-system. Specifications about the maximum rate a receiver and accept data, mechanisms to avoid network congestion,

and techniques to ensure data is received in the correct order are here. Examples of Transport layer protocols are TCP, UDP, SCTP.

- The **Application** layer specify how a pair of applications interact when they communicate, such as details about the format and meaning of messages that applications can exchange and procedures to be followed. When a programmer builds an application to communicate across a network, the programmer is devising a layer 5 protocol. Specifications for file transfer (FTP), email exchange (SMTP, POP3, IMAP), web browsing (HTTP and DNS), voice telephone service, smartphone apps, and video teleconferencing are here.

Therefore, an application on one end-system can behave as though there were a direct connection with an application running on a different end system.

The TCP/IP protocol suite operates at the top three layers. The lower layers operate with a different protocol suite (e.g. Ethernet). A router does not know about the application or transport layers.

Some of the protocols in the TCP/IP suite include the following:

- Application layer: HTTP, SMTP, DNS, DTP, POP3
- Transport layer: TCP, UDP, SCTP
- Network layer: IP, IGMP, ICMP, ARP

The range of protocols encompassed in the TCP/IP suite is very wide and is still evolving.

We will examine some of the protocols – in particular, HTTP, TCP and IP, in more detail.

TCP

When an application on one end-system sends data to another end-system, the application is controlled by an application-layer protocol. The protocol transmits data to the transport layer, where the **transmission control protocol (TCP)** operates. The TCP protocol is responsible for ensuring the safe delivery of data to the receiver. It creates packets to hold all the data, where each packet consists of the header plus the user data.

TCP also ensures that any response is sent back to the application protocol. For example, one item in the header of the data is the port number that identifies the application layer protocol (e.g. 80 for HTTP). The packet must also include the port number for the application layer protocol at the receiving end-system. If the packet is one of a sequence, the header must also include the sequence number to ensure that the data is correctly assembled by the receiving end-system.

The TCP protocol is connection-oriented. It establishes an end-to-end connection between two host computers using a **three-way handshake**. The communication goes roughly like this:

- The sender sends a packet which includes the synchronization sequence bits so that all packets will be received in the correct order.
- The receiver responds by sending back a packet containing an acknowledgement with its own synchronization sequence bits.
- The sender sends an acknowledgement that it received the receiver's packet
- The transmission from sender to receiver can now take place.

TCP uses **Positive Acknowledgement with Retransmission (PAR)**, meaning that it automatically re-sends a packet if it has not received a positive acknowledgement after a certain time interval. This allows missing packets to be identified and re-sent.

However, TCP is not concerned with the address of the receiving end-system.

IP addressing

An IP address is used to define where and to data is being transmitted. The aim is to assign a unique, universally recognised address for each device connected to the Internet.

Currently the Internet functions with **IP version 4 (IPv4)** addressing, which is based on 32 bits (four bytes) being used to define an IPv4 address. These 32 bits allow 2^{32}, or approximately 4.3 billion different addresses. IPv4 was devised in the late 1970s, before the advent of the PC and, later, smartphones and other networked devices, and therefore did not anticipate the sudden growth in the number of networked devices. In fact, not all 4.3 billion addresses are available to be used. As the number of internet users in the world grows, and as each person has multiple devices, the system will become inadequate very soon.

The original system was designed as a **hierarchical** address with a group of bits defining a network (a **netID**) and another group of bits defining a host on the network (a **hostID**).

For example, consider the IP address below. The first 16 bits are the netID and the next 16 bits are the hostID.

$$\underbrace{10111110\ 00001111}_{\text{netID}}\ \underbrace{0001100111110000}_{\text{hostID}}$$

As 32 bits is very long, it is common to abbreviate the IP address using **dotted decimal** notation. Each set of 8 bits (1 byte) is converted into its denary equivalent.

$$\underbrace{10111110}_{190}\ \underbrace{00001111}_{15}\ \underbrace{00011001}_{25}\ \underbrace{11110000}_{240}$$

The IP address above would thus be abbreviated as 190.15.25.240. The netID is 190.15 and the hostID is 25.240.

Networks are split into five different classes, as shown below. The class identifier is the first few (1 to 4) bits of the netID. This also tells you how long the netID is.

Network class	IPv4 range	Class identifier	Number of remaining bits in netID	Number of bits in hostID	Type of network
A	0.0.0.0 to 127.255.255.255	0	7	24	Very large
B	128.0.0.0 to 191.255.255.255	10	14	16	Medium
C	192.0.0.0 to 223.255.255.255	110	21	8	Small
D	224.0.0.0 to 239.255.255.255	1110	–	–	Multi-cast
E	240.0.0.0 to 255.255.255.255	1111	–	–	Experimental

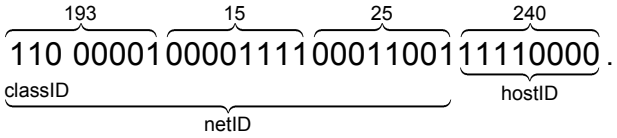
For example, a class A network IP address would be 29.68.0.43.

$$\overbrace{\underbrace{0\ 0011101}_{\text{classID}}}^{29}\ \overbrace{01000100}^{68}\ \overbrace{00000000}^{0}\ \overbrace{00101011}^{43}.$$

netID

hostID

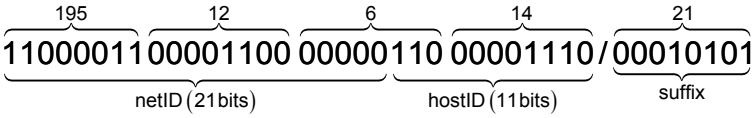
A class C network IP address would be 193.15.25.240.



Notice that a class A network can have 2^{24} = 16,777,216 possible hosts whereas a class C network can only have 2^8 = 256 possible hosts.

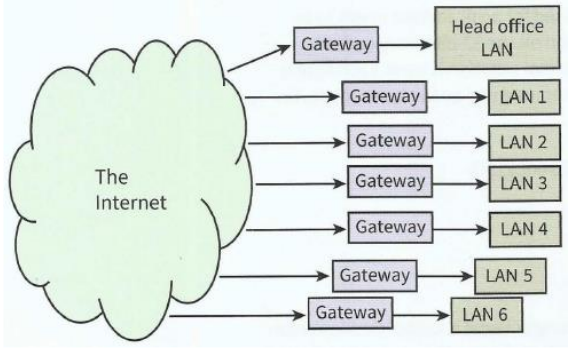
This system does not permit a lot of flexibility. For example, a network with only 300 hosts needs to be classified as a class B network, which actually allows for 2^{16} = 65,536 hosts. There will be many potential IP addresses left unused.

To address this problem, a system called **classless inter-domain routing (CIDR)** was developed to increase flexibility. A suffix is used with the IP address to indicate how many bits are used for the netID. For example, the IP address 195.12.6.14/21 means that the first 21 bits are the netID.



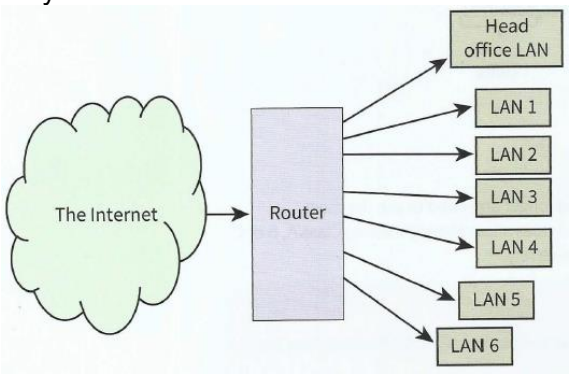
Another possible way to improve efficency in IP address allocation is to use **sub-netting**. Consider an organization with about 150 computers in 7 LANs.

If each LAN connects to the internet using its own gateway, it would look like this:



The 150 computers would be spread out among 7 class C networks. As each class C network can have 256 IP addresses, there are a total of 1792 different IP address available, but only 150 are used.

If, instead, the entire organization was allocated just one class C netID, it would only have 256 IP addresses.



Within the organization, it would have to work out how to allocate these 256 IP addresses between the 7 LANs. (One solution is to have the first three bits of the hostID indicate the number of the LAN and the remaining five bits identify the computer in the LAN.)

A third solution is called **network address translation (NAT)**. This connects an intranet to the Internet using a NAT box, which has only one IP address which is visible over the internet so it can be used as a sending address or receiving address.

Internally, the IP addresses in the intranet come from ranges of IP addresses which are reserved for private networks. Each of these addresses occurs once in a network, but can be

used simultaneously by other private networks. There is no knowledge of this on the Internet or any other private network.
The agreed ranges for private networks are 10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.31.255.255, and 192.168.0.0 to 192.168.255.255. (You do not need to memorise this.)

In time to come, however, the number of devices connected to the Internet will ultimately increase beyond the 4.3 billion unique addresses available under IPv4. A new system, **IPv6**, is being developed to increase the number of available addresses. This is a 128 bit system, so there are $2^{128} = 3.4 \times 10^{38}$ possible addresses. Because of their length, the 128 bits are broken into eight 16-bit chunks, and each chunk is converted into a 4-digit hexadecimal number. An example of an IPv6 address is (notice the colon (:) being used as a separator instead of the dot (.))

A8FB:7A88:FFF0:0FFF:3D21:2085:66FB:F0FA.

DNS

When we access a webpage or an email box, we are actually receiving data another device via the Internet. This device would have its own IP address. However, we (humans) do not want to remember many individual IP addresses using the dotted decimal system. Therefore, in 1983, the **domain name system (DNS)** was introduced, which allocates human-readable domain names for Internet hosts and provides a system for finding the IP address for a given individual domain name.

The system is stored as a hierarchical distributed database which is installed on a large number of **domain name servers** covering the entire Internet. The domain name servers are connected in a hierarchy, with powerful replicated root servers at the top of the hierarchy supporting the entire Internet. The **DNS name space**, the set of possible names, is divided into non-overlapping zones. Each zone has a primary name server with the database stored on it, and secondary servers get information from the primary server.

There are more than 250 top-level domains which may represent countries (.sg, .my, .uk) or generic organizations (.com, .org, .edu, .gov).

The domain is named by the path upward from it. For example, acjc.moe.edu.sg refers to the .acjc subdomain within the .moe subdomain in the .edu domain of the .sg top-level domain. The domain name is part of a **universal resource allocator (URL)** which identifies a webpage, or an email address.

When a domain name is typed into a web browser, the following steps take place.

1. The web browser asks the DNS server for the IP address of the website.
2. If the domain is under the jurisdiction of that server, then the correct IP address can be sent back to the user's computer.
3. If it is not under the server's jurisdiction, it may be in the server's cache of recently requested IP addresses. The IP address can still be retrieved and sent back to the user's computer.
4. If not, the DNS server sends out a request a root server, which provides an address for a DNS server with jurisdiction over the top-level domain, which can provide an address for a DNS server for the next level domain, and so on, until a server which can provide the IP address is found.
5. The first DNS server adds the IP address and associated URL into its cache, and sends it to the user's computer.
6. The user's computer communicates with the website server and the required pages are downloaded and displayed on the web browser.

Client-server architecture

In the 1980s, the traditional architecture of a mainframe computer with connected terminals was still in common use. As PCs became more common, the **client-server architecture** was developed, in which networked PCs (the clients) had access to one or more devices acting as servers.

The essence of the client-server architecture as it was first conceived is a distributed computer system where a client carries out part of the processing and a server carries out another part. In order for the client and server to cooperate, software called **middleware** has to be present. This basic concept still holds in present-day client-server applications but the language used to describe how they operate has changed.

A simple example would be a shared printer. In this case, the printer plays the role of a server (the **print server**) and the other computers are clients which send requests to the printer to be carried out – in this case, printing documents.

A summary of the interaction between the client and server is shown below.

Server Application	Client Application
Starts first	Starts second
Does not need to know which client will contact it	Needs to know which server to contact
Waits passively for contact from a client	Initiates contact when communication is needed
Communicates with client by sending and receiving data	Communicates with server by sending and receiving data
Continues to run after servicing one client, and waits for next client	Can terminate after interacting with server

Most instances of applications that follow the client-server paradigm have the following general characteristics.

Server software	Client software
Consists of a special-purpose, privileged program dedicated to providing a service	Consists of an arbitrary program that becomes a client temporarily whenever remote access is needed
Is invoked automatically when a system boots, and continues to execute through many sessions	Is invoked directly by a user, and executes for only one session
Runs on a dedicated computer system	Runs locally on a user's device
Waits passively for contact from arbitrary remote clients	Actively initiates contact with a server
Can accept connections from many clients at the same time but (usually) offers only one service	Can access multiple services as needed, but only contacts one remote server at a time
Requires powerful hardware and sophisticated operating system	Does not require especially powerful hardware

The server is now a web server which is a suite of software that can be installed on virtually any computer system. A web server provides access to a web application. The client is the web browser software. The middleware is now the software that supports the transmission of data across a network together with the provision for scripting.

It is worth emphasising that the original uses of the web involved a browser displaying web pages which contained information. There was provision for downloading of this information but the web pages were essentially static. For a client-server application, the web page is **dynamic** which means that what is displayed is determined by the request made by the client. In this context, there is almost no limit to the variety of applications that can be supported. The only requirement is that the application involves user interaction.

The most obvious examples of a client-server application can be categorised as e-commerce where a customer buys products online from a company. Other examples are: e-business, email, searching library catalogues, online banking or obtaining travel timetable information. Most applications require a web-enabled database to be installed on the server or accessible from the server.

Thin and Thick Clients

The client-server model offers **thin clients** and **thick clients**. These refer to both hardware and software.

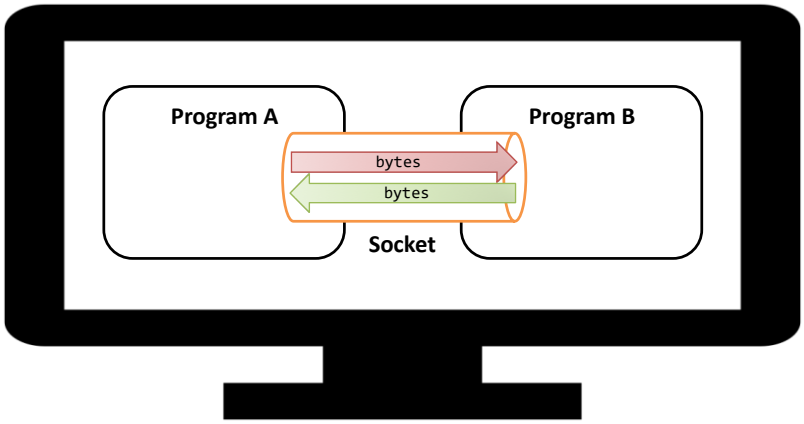
	Thin client	Thick client
Description	Heavily dependent on having a server to allow constant access to files and allow applications to run uninterrupted Needs to be connected (via LAN/WAN or Internet) to a powerful computer or server to allow processing to take place, otherwise it will not work Examples: • Web browser • POS terminal at supermarket that needs to be connected to the server to find prices, charge customers, etc	Can work offline or online, still able to do processing whether it is connected to server or not Examples: • Normal PC/laptop/tablet • Computer game that can run independently or online
Hardware advantages	Less expensive to expand (low-powered and cheap devices can be used) All devices are linked to a server (data updates and new software installation done centrally) Server offers protection against hacking and malware	More robust (device can carry out processing even when not connected to server) Clients have more control (they can store their own programs and files)
Hardware disadvantages	High reliance on the server – if the server goes down or if there is a break in communications, the devices cannot work Despite cheaper hardware, the start-up costs are generally higher than for thick clients	Less secure (relies on clients to keep their own data secure) Each client needs to update data and software individually

		Data integrity issues, since many clients access the same data which leads to inconsistencies
Software	Always relies on a connection to remote server or computer to work Requires very few local resources (such as SSD, RAM or computer processing time) Relies on good, stable and fast network connection to work Data is stored on remote server or computer	Can run some features of the software even when not connected to a server Relies heavily on local resources More tolerant of a slow network connection Can store data on local resources

2021 JC2 H2 Computing 9569
30. Socket Programming

Introduction

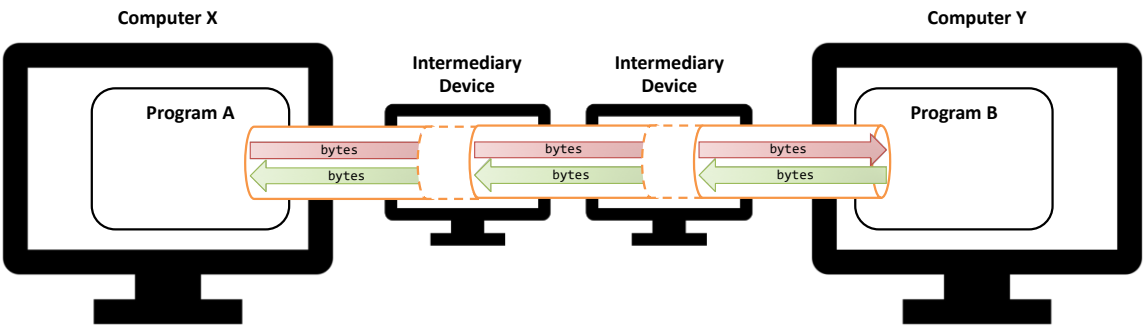
Suppose we have two Python programs running at the same time. How can we send data from one program to the other and vice versa? Most operating systems provide a powerful mechanism to do this called **sockets**.



We can picture a socket connection as a pipe between two running programs. The pipe is bidirectional and can carry data, represented by bytes, in both directions.

There are many kinds of sockets, but the kind that is most often discussed is called an **Internet socket**. Internally, Internet sockets deliver data using the same **Transmission Control Protocol and Internet Protocol suite (TCP/IP)** that is used to transmit data over the Internet. This means that Internet sockets can deliver data between *any* two programs, even programs that that are running on different computers, as long as the two computers can access each other over the network.

In reality, however, data that are transmitted through an Internet socket may pass through multiple devices before reaching the destination. Any of these devices can steal or modify the data that passes through a socket unless we encrypt the data first. An illustration of sockets that shows how the data pass through multiple devices is shown below.

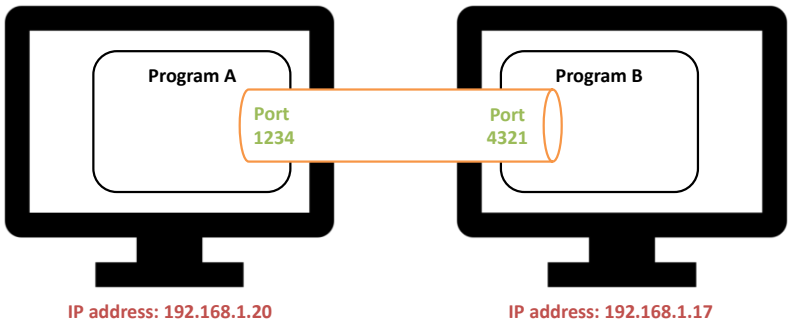


As networks can become congested, we cannot assume that data sent over Internet sockets will be transmitted instantaneously. For instance, a program may receive only the first half of

a message before the second half arrives some time later. To avoid working with incomplete data, we will need to define a **protocol** so that the start and end of messages can be detected unambiguously.

IP Addresses and Port Numbers

Each end of a socket is associated with a running program and is uniquely identified by a combined **IP address** and **port number**. The IP address identifies which device that end of the socket is attached to and the port number identifies which program on that device is using the socket.



Recall that there are two kinds of IP addresses in use today: **IPv4** addresses and **IPv6** addresses. Currently, IPv4 addresses are more frequently encountered than IPv6 addresses, so to simplify our discussion, we will be working with IPv4 addresses only.

Some IPv4 addresses are reserved for special use and have specific meanings. Two important special IPv4 addresses are:

- 127.0.0.1 refers to the local computer
- 0.0.0.0 refers to all IP addresses for local computer

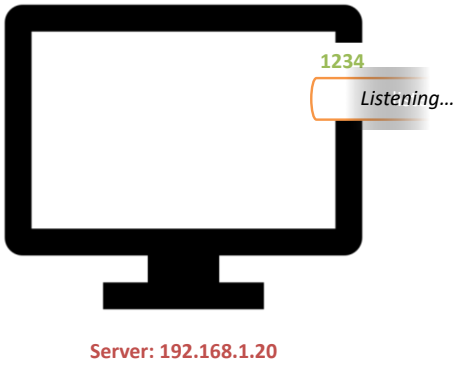
On each device, port numbers are used to distinguish between attached sockets. The device also keeps track of which program is associated with each port and which port numbers are still available for use by new sockets.

Port numbers can range from 0 to 65,535. However, the first 1,024 port numbers are reserved for specific kinds of programs and should not be used for other purposes. For instance, port 80 and port 443 are reserved for use by web server programs.

Creating a Socket Connection

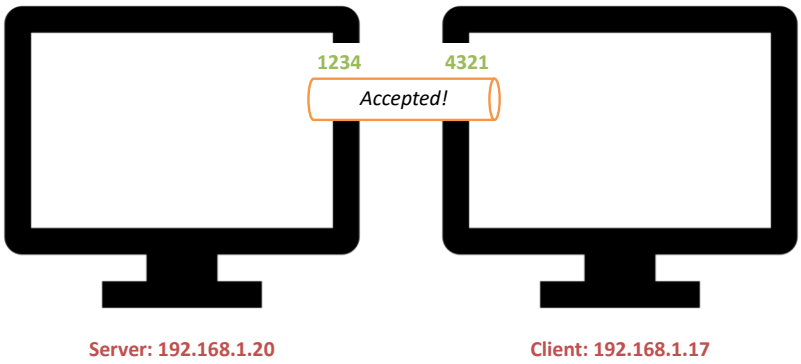
Creating a socket connection is a multi-step process that requires one program to be the **server** and another program to be the **client**. The server's IP address and port number for accepting connections must also be known ahead of time by the client.

First, the server creates a **passive socket**, binds it to the pre-chosen port number and listens for an incoming connection. A passive socket is not connected and merely waits for an incoming connection.

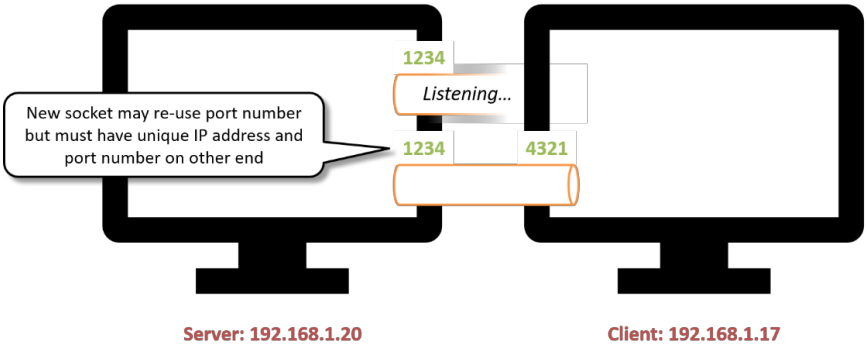


Next, the client initiates a connection request using the server's IP address and port number. If no server is listening on the chosen port, the connection will be refused.

On the other hand, if the connection request reaches an IP address and port number that a server is listening on, the server accepts and creates a new socket for the requesting client using a dynamically assigned port number.



The passive socket goes back to listening for new connections while the client and server can now exchange data using the newly-created socket.



Note that the newly-created socket is **symmetrical**: data sent on one end is received on the other end and vice versa. Once a socket is established, it can send data both from the client to the server and vice versa.

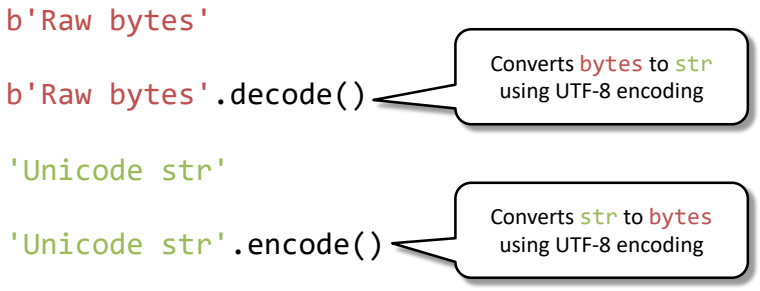
Unicode and Encodings

Before we can start writing Python code to create our own sockets, we need understand that socket work at a very basic level, so they can only send and receive data in the form of raw bytes. In other words, we must be able to encode the data into a sequence of 8-bit characters using Python's `bytes` type.

Thankfully, a Python `str` can be easily converted into `bytes` using the `str.encode()` method and vice versa using the `bytes.decode()` method.

This encoding and decoding is necessary as internally, a Python `str` is actually treated as a sequence of numbers called Unicode **code points**. There are over a million possible code points, so it is not always possible to represent each code point using just 8 bits. Instead, the Unicode standard defines an encoding called **UTF-8**, so code points can be represented using bytes in a space-efficient and consistent manner.

To enter a sequence of bytes directly in code, we can use a `bytes` literal that starts with the letter `b`, followed by a sequence of `bytes` (in the form of ASCII characters) enclosed in matching single or double quotation marks. Note that most escape codes that work for `str` literals also work for `bytes` literals.



Using the socket Module

The methods of the `socket` class are summarised in the table below.

Method	Description
`bind((host, port))`	Binds socket object to the given address `tuple`, where `host` is an IPv4 address and `port` is a port number
`listen()`	Enables socket to listen for incoming connections from clients
`accept()`	Waits for an incoming connection and returns a `tuple` containing a new socket object for the connection and an address tuple `(host, port)`, where `host` is the IPv4 address of the connected client and `port` is its port number
`connect((host, port))`	Initiates a connection to the given address tuple `(host, port)`, where `host` is the IPv4 address of the server and `port` is its port number
`recv(max_bytes)`	Receives and returns up to the given number of bytes from the socket
`sendall(bytes)`	Sends the given bytes to the socket

We can now create a basic server program. For example, let the program listen for a client on port 12345, accepts a connection request, sends `b'Hello from server\n'` to the client through the socket and finally closes the socket.

Program 1: `basic_server.py`

```
1   import socket
2
3   my_socket = socket.socket()
4   my_socket.bind(('127.0.0.1', 12345))
5   my_socket.listen()
6
7   new_socket, address = my_socket.accept()
8   print('Connected to: ' + str(address))
9   new_socket.sendall(b'Hello from server\n')
10
11  new_socket.close()
12  my_socket.close()
```

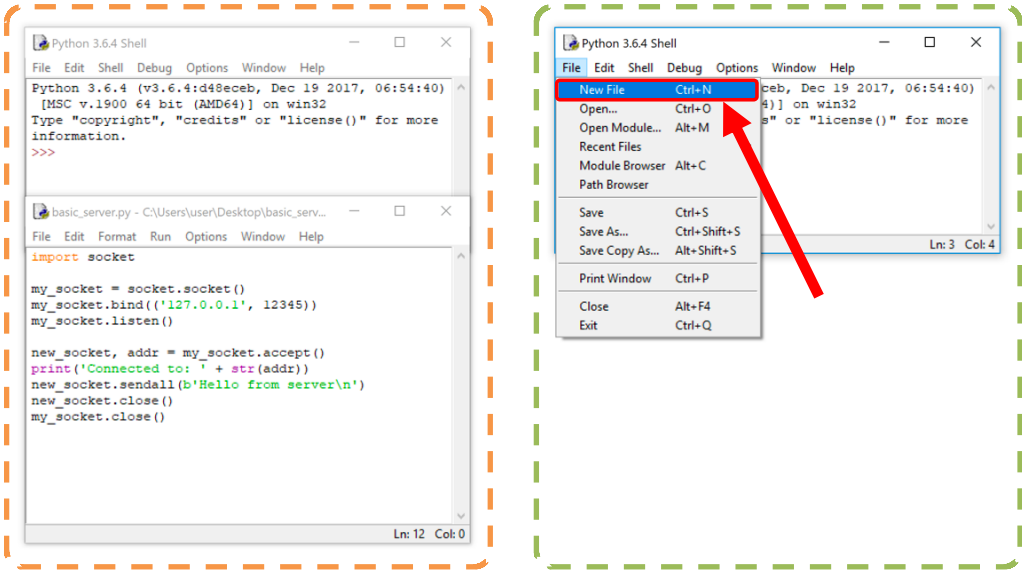
On line 7, `socket.accept()` returns a `tuple` of the newly created socket and a nested address `tuple`. We store both the new socket and the address tuple in two variables named `new_socket` and `address` respectively. Note that `new_socket` is the socket that we actually use to send and receive data.

If everything is working correctly, the server should appear stuck shortly after it is started. This is because the `socket.accept()` method is blocking[1] the program and prevents it from continuing until a connection request is received.

To create a client that can connect to this server, start a second copy of Python. For instance, if we use IDLE on Windows, open the Start Menu and run IDLE again. Move any windows from the first copy of Python to one side so the two copies of Python are clearly separated.

Copy 1 of Python Copy 2 of Python

Create a new Python program using the second copy of Python. If we use IDLE, select "New File" using the shell window that is not running the server.



Copy 1 of Python Copy 2 of Python

[1] A “blocked” process means that it is waiting for an event to occur.

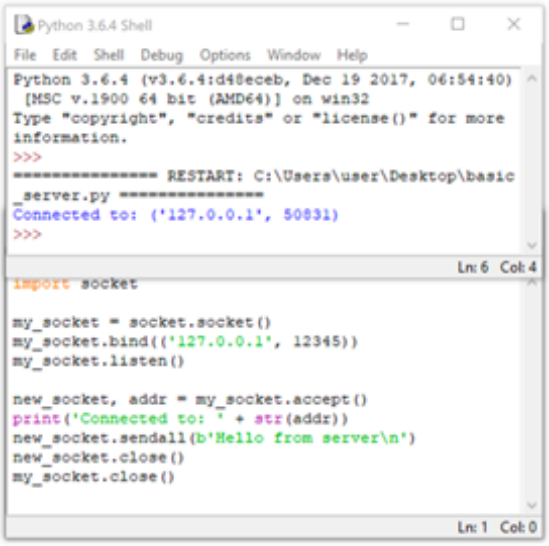
We shall now create the following basic client program that asks for the server's IP address and port number, requests for a connection, receives and prints at most 1024 bytes from the server and finally closes the socket.

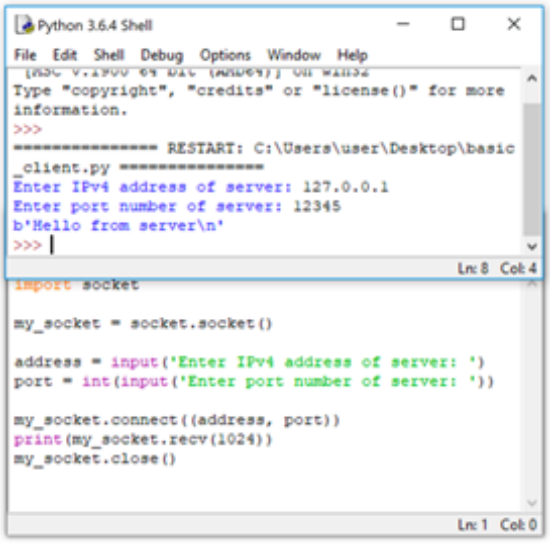
Program 2: `basic_client.py`

```
1   import socket
2
3   my_socket = socket.socket()
4
5   address = input('Enter IPv4 address of server: ')
6   port = int(input('Enter port number of server: '))
7
8   my_socket.connect((address, port))
9   print(my_socket.recv(1024))
10
11  my_socket.close()
```

On line 9, the argument for `socket.recv()` is required and should be set to a relatively small power of 2. In this case, we use a value of 2^{10} or 1024. For more information, see: https://docs.python.org/3/library/socket.html#socket.socket.recv

Run this program using the second copy of Python, ensuring that the server started previously is still running. At this point, the client should be prompting for the address and port number of the server. Use the special IPv4 address 127.0.0.1 that refers to the local machine and enter 12345 as the port number. The client should successfully connect to the server and print out the bytes that were received. At the same time, the server program should become unstuck and end normally.





Quick Check

Modify the code to demonstrate that data can be sent in the opposite direction. The client should send `b'Hello from client\n'` to the server and the server should print out any bytes that are received from the client.

Designing a Protocol

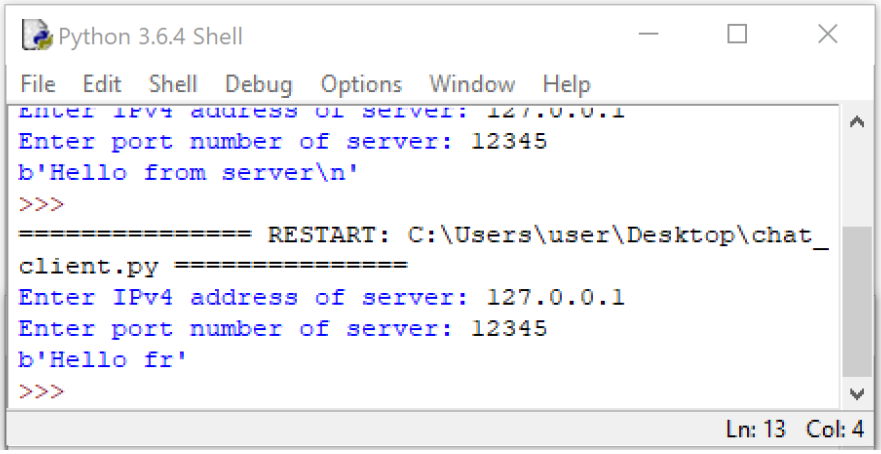
The two programs from the previous section have a hidden flaw: when using the basic server program to send longer sequences of `bytes`, only part of the data may be successfully transmitted even if we increase the maximum number of bytes that `socket.recv()` can receive.

To understand why, suppose that the sequence of `bytes` being sent is long enough that it needs to be sent as multiple packets. We can simulate this by breaking the sequence into two pieces and calling `socket.sendall()` twice, once for each piece. To simulate a busy network that may delay transport of the second packet, we also import the `time` module and call `time.sleep()` before sending the second piece.

Program 3: `basic_server_split.py`

```
1   import socket, time
2
3   my_socket = socket.socket()
4   my_socket.bind(('127.0.0.1', 12345))
5   my_socket.listen()
6
7   new_socket, address = my_socket.accept()
8   new_socket.sendall(b'Hello fr')
9   time.sleep(0.1)
10  new_socket.sendall(b'om server\n')
11
12  new_socket.close()
13  my_socket.close()
```

Run this version of the server, then run the client such that both programs run simultaneously on the same machine. This time, the client should receive only the first piece of data. If the client has closed the socket, the server may also produce an error when trying to send the second piece of data.



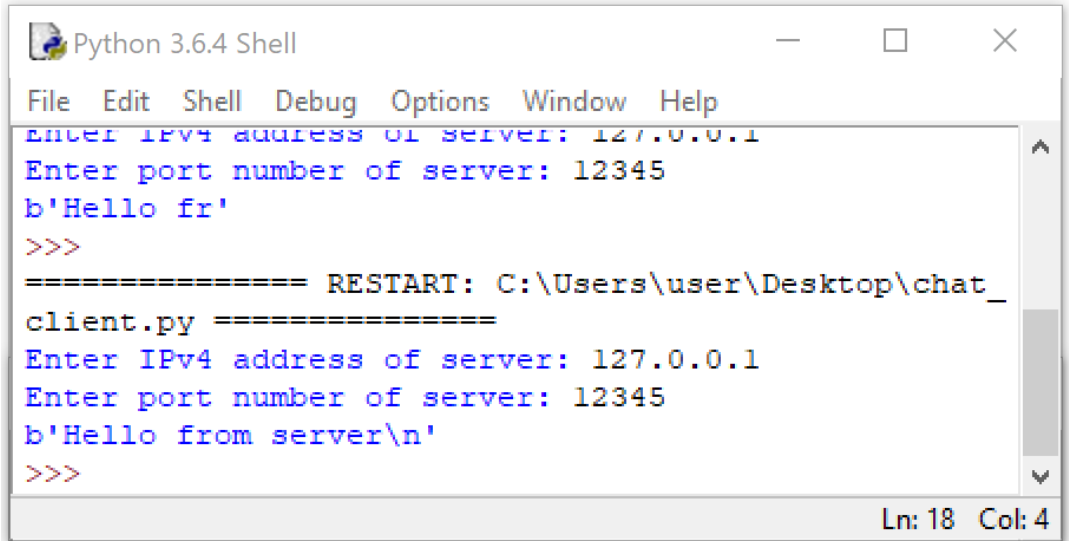
This example illustrates that, in general, we should never assume that `socket.recv()` will receive all the bytes that were sent over at one go. The only way to be certain that any received data is complete is to agree beforehand on a **protocol** or set of rules for how communication should take place. For instance, we can agree beforehand that any data we transmit will always end with a newline character `\n` and that the data itself will never contain the `\n` character. This very simple protocol allows us to detect the end of a transmission easily by just searching for the `\n` character.

The following updates the client so that it uses the `\n` character to detect when the message ends. This new client calls `socket.recv()` continuously and appends the received bytes to a variable named `data` until the `\n` character is encountered.

Program 4: `basic_client_protocol.py`

```
1   import socket
2
3   my_socket = socket.socket()
4
5   address = input('Enter IPv4 address of server: ')
6   port = int(input('Enter port number of server: '))
7
8   my_socket.connect((address, port))
9
10  data = b''
11  while b'\n' not in data:
12      data += my_socket.recv(1024)
13  print(data)
14
15  my_socket.close()
```

With this new client, all the data sent by the server up to and including the `\n` character is successfully received and printed.



Iterative and Concurrent Servers

Currently, the server program exits immediately after it finishes working with a client. In reality, we often want the server program to run continuously so that it is always listening and available for multiple clients to send connection requests. We can do this by putting the code that deals with a client in an infinite loop.

Program 5: `basic_server_iterative.py`

```
1   import socket
2
3   my_socket = socket.socket()
4   my_socket.bind(('127.0.0.1', 12345))
5   my_socket.listen()
6
7   while True:
8       new_socket, addr = my_socket.accept()
9       new_socket.sendall(b'Hello from server\n')
10      new_socket.close()
```

To interrupt a program that is running in an infinite loop, press Ctrl-C. In IDLE, we can also restart the shell using Ctrl-F6.

Internally, the server's passive socket keeps a queue of connection requests that have been received. A request is removed from this queue each time `socket.accept()` is called to create a connection. If the queue is empty, `socket.accept()` will block the program until a connection request is received, as expected.

Since `socket.accept()` is called each time the infinite loop repeats, our program is able to handle multiple clients by processing them one at a time. This means that our program works as an **iterative server**. Iterative servers are easy to write, but limited as they can only handle one client at a time.

Alternatively, we could have written our server such that it starts a **thread** that runs simultaneously with the main program each time a client tries to connect. This makes the program more complicated to write, but will let it to handle multiple clients at the same time, hence making it a **concurrent server**. We will, however, only work with iterative servers at this level.

Example: Chat Program

We now have all the tools needed to write a simple chat client and server such that two users can take turns sending single lines of text to each other. One user would be running the server and the other user would be running the client.

Since each message is restricted to a single line, we can be certain that the newline character `\n` will never be part of a message. This means that we can adopt a similar protocol of using `\n` to detect the end of a message.

Let us use a different port number of 6789 and create the following chat server program that repeatedly prompts the user for some text, sends that text to the client (after encoding it into bytes), then receives and prints out the client's response.

Program 6: chat_server.py

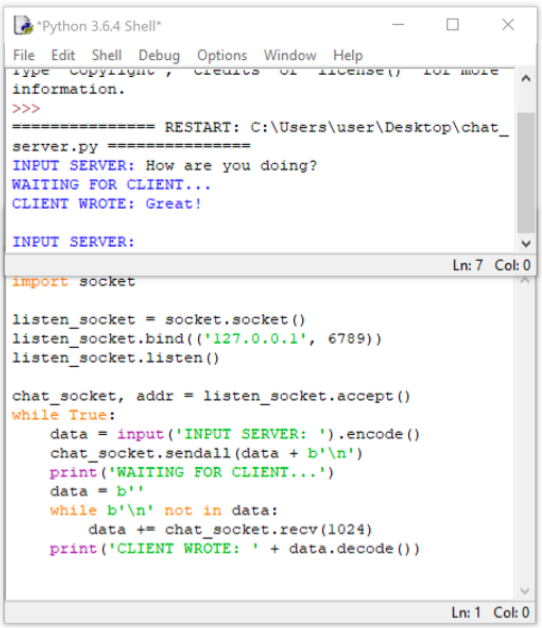
```
1   import socket
2
3   listen_socket = socket.socket()
4   listen_socket.bind(('127.0.0.1', 6789))
5   listen_socket.listen()
6
7   chat_socket, address = listen_socket.accept()
8
9   while True:
10      data = input('INPUT SERVER: ').encode()
11      chat_socket.sendall(data + b'\n')
12      print('WAITING FOR CLIENT...')
13      data = b''
14      while b'\n' not in data:
15          data += chat_socket.recv(1024)
16      print('CLIENT WROTE: ' + data.decode())
```

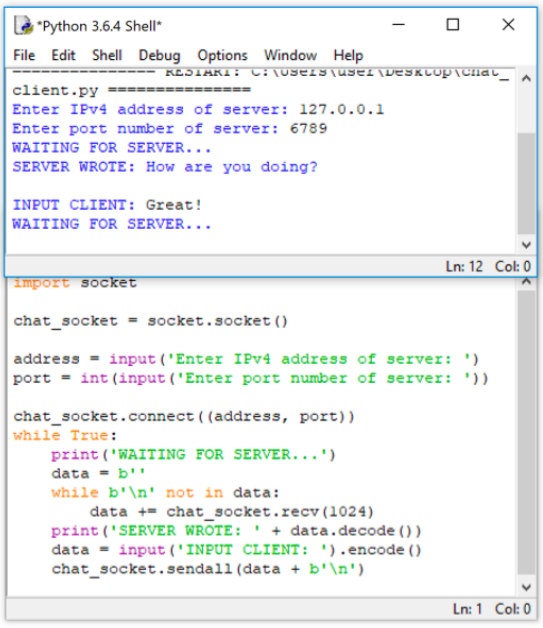
The client program is similar, except the order of sending and receiving is reversed.

Program 7: chat_client.py

```
1   import socket
2
3   chat_socket = socket.socket()
4
5   address = input('Enter IPv4 address of server: ')
6   port = int(input('Enter port number of server: '))
7
8   chat_socket.connect((address, port))
9
10  while True:
11      print('WAITING FOR SERVER...')
12      data = b''
13      while b'\n' not in data:
14          data += chat_socket.recv(1024)
15      print('SERVER WROTE: ' + data.decode())
16      data = input('INPUT CLIENT: ').encode()
17      chat_socket.sendall(data + b'\n')
```

Run the server and client using two different copies of Python. Once again, since the server is running on the same machine as the client, we can use 127.0.0.1 as the server's IPv4 address and 6789 as the port number.





Quick Check

Currently, there is no way to exit our chat programs other than to press Ctrl-C or to restart the shell (in IDLE).

Modify `chat_server.py` and `chat_client.py` so that both programs exit once the message `'quit'` is sent by any user. Ensure that all sockets are closed properly before exiting.

Example: Turn-Based Game

So far, we have been responsible for writing both the server and client programs. Sometimes, however, both server and protocol designs may be based on an existing standard or developed by someone else. To write a client that can communicate with an existing server, we need to study its code and follow the expected protocol.

Conversely, sometimes the client may be developed by someone else and we need to write a server to communicate with it. In either case, it is important to start by understanding the protocol being used.

To demonstrate how to do this, let us examine the server program for a simple turn-based 2-player game of Tic-Tac-Toe. First, we create a simple library that defines some constants and a `TicTacToe` class to handle the game logic.

Program 8: tictactoe.py

```
1   N = 3                         # Size of grid
2   WIDTH = len(str(N ** 2))      # Width for each cell
3   PLAYERS = ('O', 'X')          # Player symbols
4
5   class TicTacToe:
6
7       def __init__(self):
8           self.board = []
9           for i in range(N):
10              self.board.append([None] * N)
11
12      def render_row(self, row_index):
13          start = row_index * N + 1
14          row = self.board[row_index].copy()
15          for column_index in range(N):
16              if row[column_index] is None:
17                  cell = str(start + column_index)
18              else:
19                  cell = PLAYERS[row[column_index]]
20              if len(cell) < WIDTH:
21                  cell += ' ' * (WIDTH - len(cell))
22              row[column_index] = ' ' + cell + ' '
23          return '|'.join(row) + '\n'
24
25      def render_board(self):
26          rows = []
27          for row_index in range(N):
28              rows.append(self.render_row(row_index))
29          divider = '-' * ((WIDTH + 3) * N - 1) + '\n'
30          return divider.join(rows)
31
32      def make_move(self, player_index, cell_index):
33          cell_index -= 1
34          self.board[cell_index // N][
35              cell_index % N] = player_index
36
37      def is_valid_move(self, cell_index):
38          if cell_index < 1 or cell_index > N ** 2:
39              return False
40          cell_index -= 1
41          return self.board[cell_index // N][
42              cell_index % N] is None
43
44      def is_full(self):
45          for row_index in range(N):
46              for column_index in range(N):
47                  if self.board[row_index][
48                          column_index] is None:
49                      return False
50          return True
51
```

```
52      def get_winner(self):
53          # Check diagonals
54          if self.board[0][0] is not None:
55              found = True
56              for i in range(N):
57                  if self.board[0][0] != self.board[i][i]:
58                      found = False
59                      break
60              if found:
61                  return self.board[0][0]
62          if self.board[0][N - 1] is not None:
63              found = True
64              for i in range(N):
65                  if self.board[0][N - 1] != self.board[i][N - i - 1]:
66                      found = False
67                      break
68              if found:
69                  return self.board[0][N - 1]
70
71          # Check rows and columns
72          for i in range(N):
73              if self.board[i][0] is not None:
74                  found = True
75                  for j in range(N):
76                      if self.board[i][0] != self.board[i][j]:
77                          found = False
78                          break
79                  if found:
80                      return self.board[i][0]
81              if self.board[0][i] is not None:
82                  found = True
83                  for j in range(N):
84                      if self.board[0][i] != self.board[j][i]:
85                          found = False
86                          break
87                  if found:
88                      return self.board[0][i]
89
90          # No matching lines were found, so no winner
91          return None
```

The table below summarises the methods in `TicTacToe` class.

Method	**Description**
`render_row(row_index)`	Returns a string representation of the specified row, e.g. `1 | 2 | 3`

`render_board()`	Returns a string representation of the entire board, e.g.
	
	`1 | 2 | 3`
	`-----------`
	`4 | 5 | 6`
	`-----------`
	`7 | 8 | 9`

`make_move(player_index, cell_index)`	Modifies the board such that the specified cell is marked with the symbol for the specified player
`is_valid_move(cell_index)`	Returns whether the specified cell is currently blank
`is_full()`	Returns whether the entire board has been filled up
`get_winner()`	Returns winning player for the current board or `None` if there is no winner

Using this library, we create a server program that creates a `TicTacToe` object on line 9 to store information about the Tic-Tac-Toe board.

Program 9: `game_server.py`

```
1   import socket, tictactoe
2
3   listen_socket = socket.socket()
4   listen_socket.bind(('127.0.0.1', 3456))
5   listen_socket.listen()
6
7   game_socket, addr = listen_socket.accept()
8   game = tictactoe.TicTacToe()
9
10  while True:
11      # Display current Tic-Tac-Toe board
12      print(game.render_board())
13
14      # Check if client player won
15      if game.get_winner() is not None:
16          print('Opponent wins!')
17          print()
18          break
19
```

```
    # Check if board is full
    if game.is_full():
        print('Stalemate!')
        print()
        break

    # Prompt for move from server player
    move = -1
    while move != 0 and not game.is_valid_move(move):
        move = int(input('Server moves (0 to quit): '))
    print()
    if move == 0:
        game_socket.sendall(b'END\n')
        print('You quit, opponent wins!')
        print()
        break
    game.make_move(0, move)
    game_socket.sendall(b'MOVE' + str(move).encode() + b'\n')

    # Display current Tic-Tac-Toe board
    print(game.render_board())

    # Check if server player won
    if game.get_winner() is not None:
        print('You win!')
        print()
        break

    # Check if board is full
    if game.is_full():
        print('Stalemate!')
        print()
        break

    # Receive move from client player
    received = b''
    while b'\n' not in received:
        received += game_socket.recv(1024)
    if received.startswith(b'MOVE'):
        move = int(received[4:])
        print('Client moves: ' + str(move))
        print()
        game.make_move(1, move)
    elif received.startswith(b'END'):
        print('Opponent quits, you win!')
        print()
        break

game_socket.close()
listen_socket.close()
```

Analysing this server code, we see that communications with the client is divided into several steps that repeat in an infinite loop:

1. Display current Tic-Tac-Toe board.
2. Check if opponent has won, and if so, end game with opponent winning.
3. Check if the board is full, and if so, end game with a stalemate.
4. Prompt for input from player; if player makes a valid move, update game board accordingly, then send `b'MOVE'` followed by the chosen cell number and `b'\n'` to the opponent; if player chooses to quit, send `b'END\n'` to the opponent and end game with the opponent winning.
5. Display current Tic-Tac-Toe board again.
6. Check if player has won, and if so, end game with player winning.
7. Check if the board is full, and if so, end game with a stalemate.
8. Receive opponent's action via the socket; if the action is `b'MOVE'` followed by a cell number and `b'\n'`, update game board accordingly; if the action is `b'END\n'`, end game with the player winning.

As written, the server player always starts first. This means that our client code should start by receiving and processing the server's result. We also know that Tic-Tac-Toe is a symmetrical game (other than the choice of starting player), so we deduce that the client code should be similar to the server code except that "client" and "server" are exchanged and the last step is moved to the front.

1. Receive opponent's action via the socket; if the action is `b'MOVE'` followed by a cell number and `b'\n'`, update game board accordingly; if the action is `b'END\n'`, end game with the player winning
2. Display current Tic-Tac-Toe board.
3. Check if opponent has won, and if so, end game with opponent winning.
4. Check if the board is full, and if so, end game with a stalemate.
5. Prompt for input from player; if player makes a valid move, update game board accordingly, then send `b'MOVE'` followed by the chosen cell number and `b'\n'` to the opponent; if player chooses to quit, send `b'END\n'` to the opponent and end game with the opponent winning.
6. Display current Tic-Tac-Toe board again.
7. Check if player has won, and if so, end game with player winning.
8. Check if the board is full, and if so, end game with a stalemate.

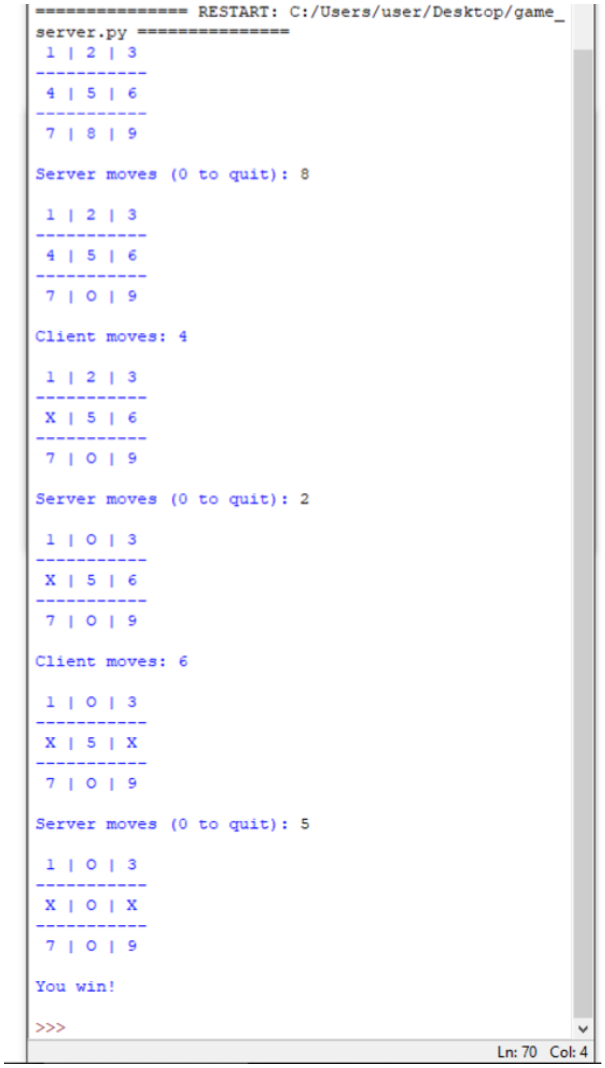
A client program that does this is as follows.

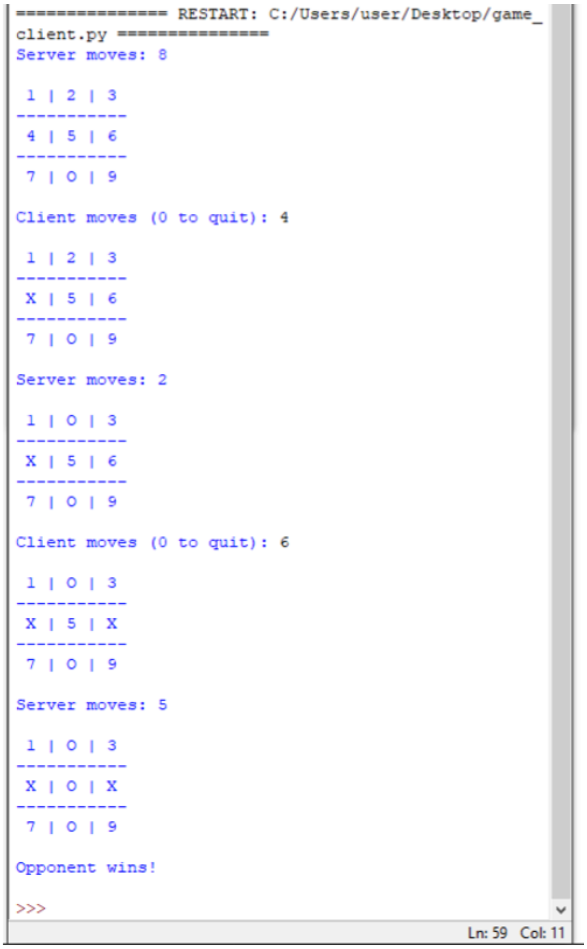
Program 10: `game_client.py`

```
1   import socket, tictactoe
2
3   game_socket = socket.socket()
4   game_socket.connect(('127.0.0.1', 3456))
5   game = tictactoe.TicTacToe()
6
7   while True:
8       # Receive move from server player
9       received = b''
10      while b'\n' not in received:
11          received += game_socket.recv(1024)
12      if received.startswith(b'MOVE'):
```

```
13          move = int(received[4:])
14          print('Server moves: ' + str(move))
15          print()
16          game.make_move(0, move)
17      elif received.startswith(b'END'):
18          print('Opponent quits, you win!')
19          print()
20          break
21
22      # Display current Tic-Tac-Toe board
23      print(game.render_board())
24
25      # Check if server player won
26      if game.get_winner() is not None:
27          print('Opponent wins!')
28          print()
29          break
30
31      # Check if board is full
32      if game.is_full():
33          print('Stalemate')
34          print()
35          break
36
37      # Prompt for move from client player
38      move = -1
39      while move != 0 and not game.is_valid_move(move):
40          move = int(input('Client moves (0 to quit): '))
41      print()
42      if move == 0:
43          game_socket.sendall(b'END\n')
44          print('You quit, opponent wins!')
45          print()
46          break
47      game.make_move(1, move)
48      game_socket.sendall(b'MOVE' + str(move).encode() + b'\n')
49
50      # Display current Tic-Tac-Toe board
51      print(game.render_board())
52
53      # Check if client player won
54      if game.get_winner() is not None:
55          print('You win!')
56          print()
57          break
58
59      # Check if board is full
60      if game.is_full():
61          print('Stalemate')
62          print()
63          break
64
65  game_socket.close()
```

Run the server and client using two different copies of Python on the same machine to verify that the game works as expected. A sample run is also provided below.





2021 JC2 H2 Computing 9569
31. Web Applications

Introduction

Web applications are programs that run in web browsers. Examples include webmails (e.g. Gmail and Hotmail), Google Docs, Youtube video player and PythonTutor.

Native applications, on the other hand, are programs that are targeted for specific platforms. For example, Youtube for Android is only for Android devices and cannot run on their Apple counterparts. Microsoft Office for Windows cannot run on Linux.

The table below shows advantages and disadvantages of web applications.

Advantages	Disadvantages
No installation required	Users need to be connected to the Internet (at least for the first instance)
The application can run on any platform that has a web browser	Since web applications are not customized for any platform, user experience may not be ideal
Easily shared among users via URL	Web applications may not have access to all device features (e.g. GPS and camera)
Easier maintenance and update as it makes use of a single codebase	Single codebase may not work well across the different browsers (e.g. Google Chrome and Safari) or even different versions of the same browser

The table below shows advantages and disadvantages of native applications.

Advantages	Disadvantages
May not require connection to the Internet (though some native applications do for complete functionality)	Users need to download native applications and install them
As native applications are customised for specific platforms, they are typically faster and/or more efficient	Users need to install updates (or allow auto-updates)
Native applications have easier access to device features	Higher development and distribution costs as every platform requires different native applications

The syllabus requires us to build simple web applications using Python with Flask as a framework, SQL, HTML and CSS.

Flask

Flask is a web application framework that allows us to use Python to serve up web pages. While HTML and CSS are used to format and beautify web pages respectively, a programming language operates in the background to process user requests.

Program 1: `flask_minimal.py`

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  if __name__ == '__main__':
6      app.run()
```

Without any customisations, Flask already provides a basic web server that correctly implements **Hypertext Transfer Protocol (HTTP)**, which is the underlying format that is used to structure requests and responses for effective communication between a client and a server.

To create and run this basic web server, we need to create a `flask.Flask` object with the module's `__name__` as an argument and call the object's `run()` method.

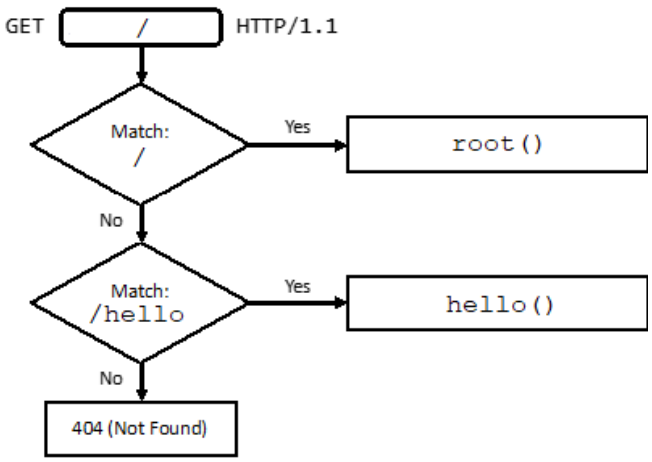
When this program is run, we should see some start-up messages that indicate the server can be accessed at `http://127.0.0.1:5000/`. However, as the default web server is not configured to recognise any paths yet, we will receive a 404 (Not Found) error when we visit that URL using a web browser.

Note that 5000 is the default port number used by Flask. To use another port number, we can call the `run()` method with a different port argument, e.g. `app.run(port=12345)`.

HTTP Requests and Routing

Simply put, **HTTP requests** are messages sent by clients to servers. One of such request is **GET**, which is used to request data from a specified resource in a server.

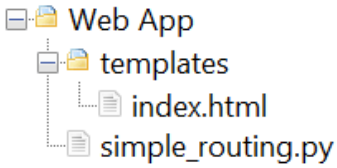
Suppose we have a simple web application with only two paths: the root path `/` and another called `/hello`. As an example, the routing process to get the data associated with the root path is summarised in the flowchart below.



Program 2: `simple_routing.py`

```
1   from flask import Flask, render_template
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       return render_template('index.html')
8
9   @app.route('/hello')
10  def hello():
11      return "Hello world!"
12
13  app.run()
```

HTML files must be placed in the folder `templates`. Create the following directory and files.



To declare a route and associate a path to a Python function, we use a feature called **decorator**, each starting with `@` as shown on lines 5 and 9.

Try accessing `http://127.0.0.1:5000/` and `http://127.0.0.1:5000/hello`.

Program 3: `complex_routing.py`

```
1   from flask import Flask, render_template
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       return render_template('index.html')
8
9   @app.route('/one')
10  @app.route('/one/two')
11  def test_multiple():
12      return "Mic test: one, two..."
13
14  @app.route('/string/<s>')
15  def string_variable(s):
16      return "You typed a string: {}".format(s)
17
18  @app.route('/integer/<int:i>')
19  def integer_variable(i):
20      return "You typed an integer: {}".format(i)
21
22  app.run()
```

Lines 9-12 show that two paths can lead to the same function.

Lines 14-20 show that Flask routes can also have variable parts. Each variable has a name surrounded by <>, whose associated data can be processed further.

There are times when we want to lead users from an outdated web page to a new one instead. This can be done via **redirection** as shown in the two programs below.

Program 4: `redirection1.py`

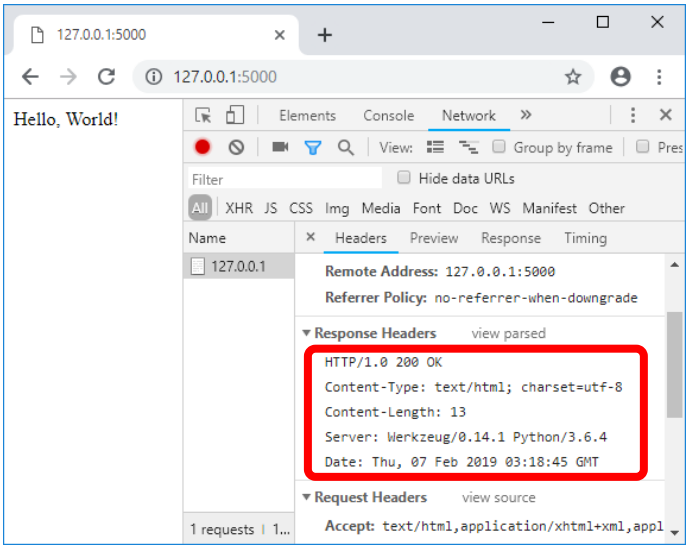
```
1   from flask import Flask, render_template, redirect, url_for
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def old():
7       return redirect(url_for('new'))
8
9   @app.route('/new')
10  def new():
11      return render_template('index.html')
12
13  app.run()
```

Program 5: `redirection2.py`

```
1   from flask import Flask, render_template, redirect
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def old():
7       return redirect("http://www.google.com")
8
9   app.run()
```

HTTP Responses and Status Codes

How do the short strings returned in some of the functions shown earlier get displayed without any issues in the web browser? The answer is that Flask actually prepends various headers behind the scenes to produce valid **HTTP responses**, which are messages sent by servers to clients.



Notice that Flask assumes that our output has a `Content-Type` of `text/html`. This means that it actually expects our function to return a full HTML document and not just a plain string. However, most browsers are very forgiving and will treat our string as a snippet of HTML intended for `<body>`.

Besides that, notice that Flask also assumes that our responses have a HTTP status code of 200 (OK) by default. This is usually what we want, but if needed, we can override this by returning a `tuple` instead of just a `string`. as demonstrated on line 7 of the Python code below.

Program 6: `status_500.py`

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  @app.route('/')
6  def index():
7      return ('', 500)
8
9  app.run()
```

A HTTP status code of 500 represents an Internal Server Error.

Processing Forms

Create the following HTML file.

HTML 1: `name_form_get.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Name Form</title>
6       </head>
7
8       <body>
9           <form action="{{url_for('show')}}" method='GET'>
10              <p>Surname: <input type="text" name="surname"></p>
11              <p>Given name: <input type="text"
                                  name="given_name"></p>
12              <input type="submit" value="Submit!">
13          </form>
14      </body>
15  </html>
```

There are two attributes in the `<form>` tag shown above.

Attribute	Purpose
`action`	Determines where the input data are submitted to
`method`	Specifies the HTTP method to be used, i.e. GET (default) or POST

Notice that there is a rather unfamiliar syntax of `{{url_for('show')}}` in the HTML code above. Such is written in **Jinja2**, a web template language that works in conjunction with Flask.

In Jinja2, `{{<statement>}}` represents print.

This particular code instructs Python to retrieve the web address associated with the `show()` function in `form1a.py` shown on the next page when the form is submitted.

Program 7: `form1a.py`

```
1   from flask import Flask, render_template, request
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       return render_template('name_form_get.html')
8
9   @app.route('/show')
10  def show():
11      surname = request.args['surname']
12      given_name = request.args['given_name']
13      return render_template('show1.html', name1=surname,
                               name2=given_name)
14
15  app.run()
```

On lines 11-12, `request.args` are `dictionary`-like objects that contain the data submitted through the form. We can access each piece of data on the query portion of the URL using the `name` of the input field in `name_form_get.html`.

We also need a second HTML file to display the input name as follows.

HTML 2: `show1.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Show Page</title>
6       </head>
7
8       <body>
9           <p>Hello, {{name1}} {{name2}}!</p>
10      </body>
11  </html>
```

On line 8, `{{name1}}` and `{{name2}}` correspond to the variable name declared in the `show()` function in `form1.py` on line 13.

Notice on the address bar that you can see the form data submitted. As such, HTTP GET has several disadvantages:

- The submitted form data are recorded in the resulting URLs, which allows anyone to view our browser history to obtain the data sent.
- Some browsers and server software limit the length of URLs, so there is a risk that overly long form data submitted using GET requests may get truncated.
- GET requests are not supposed to make changes to the server's data. If we use data submitted with GET requests to add, delete or update data from a database, we are not following the HTTP standard.

To overcome these disadvantages, we can use another HTTP request, which is **POST**. Such requests do not remain in the web browser history, cannot be bookmarked and are never cached, providing us a secure way of sending sensitive data.

Create a copy of HTML1 and change the method on line 9 to `'POST'`. Save the file as `name_form_post.html`.

We also need to modify our Python code in order for the POST request to work. The changes required are shown in bold.

Program 8: `form1b.py`

```
1   from flask import Flask, render_template, request
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       return render_template('name_form_post.html')
8
9   @app.route('/show', methods=['POST'])
10  def show():
11      surname = request.form['surname']
12      given_name = request.form['given_name']
13      return render_template('show1.html', name1=surname,
                               name2=given_name)
14
15  app.run()
```

On line 9, it is necessary to specify the POST request in the decorator.

On lines 11-12, the submitted form data using the POST request are placed in `request.form` `dictionary`-like object instead of `request.args`.

Jinja2: FOR Loops

Must we know all the keys to the data stored in the form to retrieve them? It is actually possible to loop through the keys using Jinja2.

We will still use `name_form_post.html` to collect the data, but we shall come up with a new Python program and a HTML file to show the data for this purpose.

Program 9: `form2.py`

```
1   from flask import Flask, render_template, request
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       return render_template('name_form_post.html')
8
9   @app.route('/show', methods=['POST'])
10  def show():
11      return render_template('show2.html', data=request.form)
12
13  app.run()
```

HTML 2: `show2.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Show Page</title>
6       </head>
7
8       <body>
9           <p>We can arrange the data if we know the keys.<br>
10              {{data['surname']}} {{data['given_name']}}<br>
11              OR<br>
12              {{data['given_name']}} {{data['surname']}}</p>
13
14          <p>Alternatively, we can do a FOR loop.<br>
15              {%for item in data%}
16              {{data[item]}}<br>
17              {%endfor%}</p>
18      </body>
19  </html>
```

As shown above, the Jinja2 syntax for FOR loop is as follows.

```
{%for <variable1> in <variable2>%}
...
{%endfor%}
```

Jinja2: Conditionals

Conditional statements involving if, else-if and else are supported in Jinja2.

The Python program below is similar to **Program 8** using HTTP POST, except on line 13 where `show3.html` is to be rendered instead.

Program 10: `form3.py`

```
1   from flask import Flask, render_template, request
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       return render_template('name_form_post.html')
8
9   @app.route('/show', methods=['POST'])
10  def show():
11      surname = request.form['surname']
12      given_name = request.form['given_name']
13      return render_template('show3.html', name1=surname,
                               name2=given_name)
14
15  app.run()
```

HTML 3: `show3.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Show Page</title>
6       </head>
7
8       <body>
9           <p>
10              {{name1}}<br>
11              {{name2}}<br>
12              {%if name1|length > 3%}
13              The surname is longer than three letters.
14              {%elif name1|length > 1%}
15              Surname is entered.
16              {%else%}
17              No surname is entered.
18              {%endif%}
19          </p>
20      </body>
21  </html>
```

On lines 12 and 14, `name1|length` is the syntax for obtaining the length of the string `name1`, which is the surname entered.

As shown in the example, the Jinja2 syntax for conditionals is as follows.

```
{%if <statement1>%}
...
{%elif <statement2>%}
...
{%else%}
...
{%endif%}
```

Even though it is possible to perform input validation using Jinja2 conditionals, it is typically better to do that within the Python program.

Jinja2: Displaying HTML Code

It is possible to include snippets of HTML code in a Python program as a string and get it rendered using Jinja2.

Program 11: `html_ex.py`

```
1   from flask import Flask, render_template, request
2
3   app = Flask(__name__)
4
5   @app.route('/')
6   def root():
7       html = "<b>This sentence is bolded.<b>"
8       return render_template('show_html_ex.html', code=html)
9
10  app.run()
```

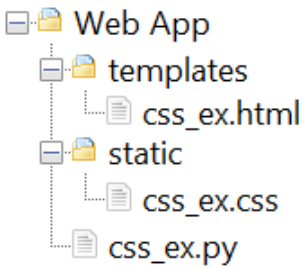
HTML 4: `show_html_ex.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Show Page</title>
6       </head>
7
8       <body>
9           <p>Without using safe: {{code}}</p>
10          <p>Using safe: {{code|safe}}</p>
11      </body>
12  </html>
```

Recall that `{{}}` represents print in Jinja2. As shown on line 9, to render the HTML code, the `|safe` filter should be used. Otherwise, it will be treated as a normal string.

Flask and CSS

Unlike HTML files that are placed in the folder `templates`, CSS files must be placed in the folder `static`. Create the following directory and files.



HTML 5: `css_ex.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Flask with CSS</title>
6           <link rel='stylesheet'
             href={{url_for('static',filename='css_ex.css')}}>
7       </head>
8
9       <body>
10          <p id="p1">This is a text.</p>
11          <p id="p2">This text must have a different style.</p>
12      </body>
13  </html>
```

Quick Check

Write `css_ex.py` and `css_ex.css`., ensuring that the two texts in the `<body>` appear differently when rendered.

File Handling

Forms may ask users to upload certain files. Uploaded files can be stored in the folder static or another folder of choice. For this purpose, we shall create the folder `uploads`.

The following HTML form requests for two files – a text file and an image file – to be submitted.

HTML 6: `file_input.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>File Input Form</title>
6       </head>
7
8       <body>
9           <form action="{{url_for('show')}}" method='POST'
             enctype='multipart/form-data'>
10              File input: <input type="file" name="file1"><br>
11              Image input: <input type="file" name="file2"><br>
12              <input type="submit" value="Submit!">
13          </form>
14      </body>
15  </html>
```

On lines 10-11, it is shown that an `<input>` tag can have `type="file"`. For file uploading to work properly, the enclosing `<form>` on line 9 must also be configured to use HTTP POST and include the additional attribute `enctype="multipart/form-data"`. This means that one or more sets of data are combined and encoded as a single body.

The following HTML code shows the first line of the text file, as well as the image file uploaded by the user.

HTML 7: `file_output.html`

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>Show Output</title>
6       </head>
7
8       <body>
9         <p>First line of the text file:<br>
             {{line1}}</p>
10        <p>Photo uploaded:
11           <img src="{{url_for('get_file', filename=photo)}}">
12        </p>
13      </body>
14  </html>
```

The following is the Python code required for this web application to work.

Program 12: `file_io.py`

```
1   import os
2   from flask import Flask, render_template, request,
                      send_from_directory
3   from werkzeug.utils import secure_filename
4
5   app = Flask(__name__)
6
7   @app.route('/')
8   def root():
9       return render_template('file_input.html')
10
11  @app.route('/show', methods=['POST'])
12  def show():
13      # Obtain the text file
14      f1 = request.files['file1']
15
16      # Decode the byte string into a normal string
17      line1 = f1.readline().decode('ASCII')
18
19      # Obtain the image file
20      photo = request.files['file2']
21      filename = secure_filename(photo.filename)
22
23      # Save the image in the specified directory
24      photo_path = os.path.join('uploads', filename)
25      photo.save(photo_path)
26
27      return render_template('file_output.html', line1=line1,
                               photo=filename)
28
29  @app.route('/uploads/<filename>')
30  def get_file(filename):
31      return send_from_directory('uploads', filename)
32
33  app.run()
```

On lines 14 and 20, we can see that submitted files are accessed from `request.files` dictionary instead of `request.form`.

In general, we should be careful whenever we let users specify filenames for reading or writing files on the server's file system as file paths can use special folder names such as, `..`, to access parent folders that may contain source code or our server's configuration files. To prevent this from happening, on line 21, we pass the filename through the `secure_filename()` function provided in the `werkzeug.utils` module first. This function returns a modified filename with any special characters replaced so that it can be safely treated like a normal filename. We then use this filename on line 24 to form a file path that is guaranteed to be in our folder `uploads`. Lines 29-31 provide the way to view the file from the route that we have established.

Flask and SQL

SQL database is very often an integral part of a web application to manage all the data supplied by users.

In this section, we shall build a web page that allows each user to key in his/her name, class and gender. The data are to be stored in an SQL database.

The HTML code is shown below.

HTML 8: `sql_form1.html`

```
1   <!DOCTYPE html>
2
3   <html>
4         <head>
5               <title>SQL Example</title>
6         </head>
7
8         <body>
9               <form action="{{url_for('store')}}" method='POST'>
10              Name:<input type="text" name="name"><br><br>
11              Class:<input type="text" name="class"><br><br>
12              Gender:<br>
13              <input type="radio" name="gender" value="M"> Male
14              <input type="radio" name="gender" value="F"> Female
15              <br><br>
16              <input type="submit" value="Submit!">
17              </form>
18        </body>
19  </html>
```

For this simple exercise, we shall assume that all names to be keyed into the database are different.

Before looking at the Python code on the next page, can you try to write it on your own to serve up the above HTML form?

Program 13: sql_ex1.py

```
1   import sqlite3
2   from flask import Flask, render_template, request
3
4   app = Flask(__name__)
5
6   @app.route('/')
7   def root():
8       return render_template('sql_ex1.html')
9
10  @app.route('/store', methods=['POST'])
11  def store():
12      name = request.form['name']
13      form_class = request.form['form_class']
14      gender = request.form['gender']
15
16      connection = sqlite3.connect("school.db")
17      connection.execute('''
18                         CREATE TABLE IF NOT EXISTS school (
19                         name TEXT PRIMARY KEY,
20                         class TEXT,
21                         gender CHAR(1)
22                         )''')
23
24      connection.execute("INSERT INTO school VALUES, (?, ?, ?)",
25                         (name, form_class, gender))
26
27      connection.commit()
28      connection.close()
29
30      return "The data have been saved."
31
32  app.run()
```

We shall now create another two web pages. One of them is to ask users to key in a name, and the other is to display the class and the gender if the name exists in the database.

HTML 9: sql_form2.html

```
1   <!DOCTYPE html>
2
3   <html>
4       <head>
5           <title>SQL Example</title>
6       </head>
7
8       <body>
9           <form action="{{url_for('show')}}" method='POST'>
10          Name:<input type="text" name="name"><br><br>
11          <input type="submit" value="Submit!">
12          </form>
13      </body>
14  </html>
```

HTML 10: sql_show2.html

```
<!DOCTYPE html>

<html>
      <head>
            <title>SQL Example</title>
      </head>

      <body>
            <b>Name</b>: {{data['name']}}<br>
            <b>Class</b>: {{data['form_class']}}<br>
            <b>Gender</b>: {{data['gender']}}
      </body>
</html>
```

The Python code is as follows.

Program 14: sql_ex2.py

```
import sqlite3
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/')
def root():
    return render_template('sql_form2.html')

@app.route('/show', methods=['POST'])
def show():
    name = request.form['name']

    connection = sqlite3.connect("school.db")
    connection.row_factory = sqlite3.Row
    cursor = connection.execute("SELECT * FROM school WHERE
                                    name = ?", (name,))
    result = cursor.fetchone()
    connection.close()

    if result == None:
        return "The student data do not exist."
    else:
        return render_template('sql_show2.html', data=result)

app.run()
```

Quick Check

Combine the two separate web applications in this section into one. In a single Python program, use the route `'/form'` `'/check'` and `'/show'` to serve up **HTML 8**, **HTML 9** and **HTML 10** respectively.

Usability Principles

We shall try to adhere to certain general principles when building applications to ensure that they are user-friendly.

1. Keep users informed of the system's status, e.g.
    - Download status bar in web browsers
    - Wi-Fi icon on your phone
    - Battery level indicator

2. Match between system and the real world, e.g.
    - Use phrases, icons and concepts understandable by users
    - E-book reader allows users to turn the page by swiping the screen from right to left (corresponding to how one flips a physical book) and also add bookmarks
    - Volume control buttons with the top button to increase volume and the bottom button to decrease the volume

3. User control and freedom, e.g.
    - Allow users to undo or redo (e.g. on web browsers or word processors)
    - Allow users to return to the previous menu or exit an application easily

4. Consistency and standards, e.g.
    - Follow conventions and use the same term to mean the same thing consistently; users should not be put in a position to guess if two terms are referring to the same thing
    - In Windows, the close window button is always at the top right-hand corner (labelled as X)
    - Most shopping websites have a shopping cart page (with a corresponding shopping cart icon) for users to review the items added before paying for them

5. Error prevention, e.g.
    - Include helpful constraints, e.g.
        - Use a calendar to accept birthday input as opposed to using a text box
        - Use a radio button to restrict users to only valid choices
    - Display confirmation dialogues, e.g.
        - "Are you sure you want to delete this record?"
        - "Do you want to exit without saving?"
    - Provide an undo button for people to prevent making an error permanent

6. Recognition rather than recall, e.g.
    - Use common icons
    - Make objects and options clearly visible
    - Shopping websites provide a section on previously bought items for users to revisit what they last bought (and probably want to buy again)
    - In Microsoft Word, there are lists of recently opened documents, common templates, etc.

7. Flexibility and efficiency of use, e.g.
    - Allow multiple ways of achieving the same result, e.g. copy and paste can be done by using Ctrl+C shortcut or using the edit menu, allowing different users to choose the method that is most convenient for them
    - In Mac OS, users have the freedom to create their custom keyboard and shortcut commands

8. Aesthetic and minimalist design, e.g.
    - Some visuals, such as vibrant colours of food and scenic landscapes with water bodies, signal promises of fulfilling human needs, such as food, water, shelter, safety, warmth, companionship an community
    - Provide only what is necessary as redundant information clutters the screen and competes with useful information, e.g. Google search engine has a very simple and clean design

9. Help users recognise, diagnose, and recover from errors, e.g.
    - Provide error messages in simple language, e.g. “Enter a valid e-mail address.”
    - Give specific and constructive advice, e.g. “Your password must contain at least one uppercase letter” is much better than “Your password does not meet the requirements”.

10. Help and documentation, e.g.
    - Ideally, any applications should be usable without any documentation
    - However, it is still a good idea to provide some documentation, as concise as possible to assist users

References

GET vs POST: https://www.w3schools.com/tags/ref_httpmethods.asp
HTML forms: https://www.w3schools.com/html/html_forms.asp
Usability principles: https://www.nngroup.com/articles/ten-usability-heuristics/

Summary

`request` **Object**

Attribute	`request.args`	`request.form`	`request.files`
Content	Dictionary of field names and their associated values from query portion of URL	Dictionary of field names and their associated values	Dictionary of file upload names and their associated `FileStorage` objects
HTTP Method	Usually GET (but also works with POST if URL has query portion)	POST only	POST only
Use	Reads form data submitted using GET	Reads form data submitted using POST	Reads files submitted using POST
`<form>` Attribute	If not specified, it is `method='GET'` by default	Must specify `method='POST'`	Must specify `method='POST' enctype='multipart/form-data'`

`flask` Module

`Flask(__name__)`	Creates a Flask application object
`render_template (filename, var=value)`	Renders Jinja2 template with the given `filename` using the given variable values
`redirect(path)`	Redirects user to the given `path` when used as a return value of a decorated function
`url_for (name, var=value)`	Returns the path that is mapped to the given function name and given variable values
`request`	Accesses current `request` object
`send_from_directory (directory, filename)`	Sends file from the given `directory` with the given `filename` when used as a return value of a decorated function

`werkzeug` Module

`secure_filename (filename)`	Replaces all characters that have special meanings (e.g. path separators) in the given `filename`

`os` Module

`path.join (foldername, filename)`	Creates a file path where the `filename` is to be inside the `foldername`

`Flask` Class

`route (path, methods=[…])`	Maps the given `path` to the decorated function with the HTTP `methods`: `'GET'` or `'POST'` or both inside the `list`.
`run()`	Runs the Flask application (with optional arguments `port=<integer>` as the port number and/or `debug=True` to enable the debug mode)

`FileStorage` Class

`filename`	Gives the name of the uploaded file
`save(path)`	Saves the uploaded file to the given `path`

32 Social, Ethical, Legal and Economic Issues

Ethics

There is no standard definition of what 'ethics' really is, but the following three sentences give an idea:

- Ethics is the field of moral science.
- Ethics are the moral principles by which any person is guided.
- Ethics are the rules of conduct recognised in a particular profession or area of human life.

For our purposes, the third definition is the most useful. Of course, these rules reflect the moral principles that come from the second definition. The following observations come to mind when considering moral principles.

Moral principles concern right or wrong. The concept of virtue is often linked to what is considered to be right. Some viewpoints for deciding whether something is right or wrong are the following:

- **Philosophical** thinkers such as Aristotle, Confucius, and others are often quoted in this context, as they have analysed in great detail why certain thoughts and actions should be considered right or wrong, and tried to distil fundamental principles for deciding morality.
- **Religious** points of view can incorporate philosophical ones, or introduce their own new ones. We will not discuss specific religious beliefs, except to point out that they do have to be considered in the working and social environment.
- **Legal** frameworks, such as the laws of a particular country, should reflect what is right and wrong, and certainly have an impact on working practices, but are rarely the primary focus in rules of conduct.
- **Pragmatic** consideration can be roughly defined as applying common sense. These, together with the philosophical view of right and wrong, usually form the foundation for creating rules of conduct.

Codes of Ethics

Organizations normally have codes of ethics to guide their members in deciding what is right and wrong. For professional organizations, they expect their members to uphold a certain moral standard, so that the reputation and integrity of the organization and the profession, are not compromised.

Two American organizations that have a strong global perspective and influence are the **Association for Computing Machinery (ACM)** and the **Institute of Electrical and Electronics Engineers (IEEE)**. Their code of ethics[1] is therefore one that is often referenced with regards to moral considerations for software engineers.

This code of ethics is **not** a look-up table that prescribes a certain action in a given situation. Rather, it is a set of fundamental principles, and they advocate that a professional software engineer should make an ethical judgement based on thoughtful considerations of these principles.

[1] https://ethics.acm.org/code-of-ethics/software-engineering-code/

The eight principles defined in the code of ethics are the following:

1. **Public** – Software engineers shall act consistently with the public interest.
2. **Client and Employer** – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. **Product** – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. **Judgment** – Software engineers shall maintain integrity and independence in their professional judgment.
5. **Management** – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. **Profession** – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. **Colleagues** – Software engineers shall be fair to and supportive of their colleagues.
8. **Self** – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Altogether there are 80 clauses. Many of them do not relate specifically to software engineering, but actually relate to proper behaviour for any group of professional workers. For example:

2.03. Use the property of a client or employer only in ways properly authorized, and with the client's or employer's knowledge and consent.

5.04. Assign work only after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.

5.05. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work, and provide an uncertainty assessment of these estimates.

6.06. Obey all laws governing their work, unless, in exceptional circumstances, such compliance is inconsistent with the public interest.

Take note, for instance, of the qualifiers in 5.04 and 6.06 ("tempered with", "unless"). These exist in many of the clauses, showing the difficulty of applying a one-size-fits-all guiding principle to all possible scenarios. 5.05 is also notable for insisting on an uncertainty assessment.

The **Singapore Computer Society** is the primary infocomm and digital media professional society in Singapore, with about 33 000 members and 16 specialist groups. Its code of conduct[2] also guides its members in their professional behaviour. The guidelines are broadly categorised into four main categories:

1. **integrity**,
2. **full responsibility**,
3. **competence**,
4. **professionalism**.

[2] https://www.scs.org.sg/membership/membership_code_of_conduct.php

Likewise, the **British Computer Society**'s code of conduct[3] also gives guidance under four headings:

1. **public interest**,
2. **professional competence and integrity**,
3. **duty to relevant authority**,
4. **duty to the profession**.

Regardless of the different details in each professional organization's code of ethics or code of conduct, there is a general consistency with regard to the following:

1. The public interest or public good is a key concern.
2. The codes present fundamental principles.
3. The professional is expected to exercise their own judgment.
4. The professional should seek advice if unsure.

The public good

So far, we have been considering professional working practices, which revolves around the third definition of ethics. When the question of public good arises, we also need to start considering the second definition as well.

The ACM/IEEE code of ethics refers to the following:

- The health, safety and welfare of the public,
- The public interest,
- The public good,
- Public concern.

Likewise, the SCS code of conduct refers to "the advancement of human welfare" while the BCS code of conduct states that the professional should "have due regard for public health, safety, privacy, security and wellbeing of others and the environment".

There is no further indication of how these should be interpreted. However, we can look at some case studies to illustrate what might be considered.

1. The Therac-25 was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982. The machine offered two modes of radiation therapy:
    a. Direct electron-beam therapy, which involved a narrow low-current beam of electrons directly hitting the patient, and
    b. Megavolt X-ray therapy, which involved a current 100 times higher, which then struck a target to produce X-rays. The X-ray beam was supposed to pass through a flattening filter and a collimator before hitting the patient at a safe level.
    c. A third mode, Field Light mode, allowed the patient to be correctly positioned by using visible light to illuminate the target area of the body.

    In previous models of the Therac, hardware design meant that if the machine was in X-ray therapy mode, the high-current electron beam would be physically blocked from

[3] https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/

reaching the patient. However, in the Therac-25, this safety feature was replaced by a software check. There were two bugs in the software:

a. When the operator incorrectly selected X-ray mode before quickly changing to electron mode, this allowed the electron beam to be set for X-ray mode without the X-ray target being in place.
b. The electron beam could activate during field-light mode, during which no beam scanner was active or target was in place.

As a result of these bugs, patients were sometimes hit with an electron beam 100 times larger than intended, delivering a potentially lethal dose of beta radiation. Between 1985 and 1987, there were at least six documented cases which resulted in three deaths.

Correct application of the code of ethics with respect to specification, development and testing of software could have saved human life. Fortunately, examples where software bugs cause loss of life are very rare indeed. In the next three case studies, if the software had been documented and tested properly, large amounts of public funds could have been saved.

2. The Ariane 5 rocket exploded 40 seconds after blast-off in 1996. About US$500 million had gone into its development, scientific equipment and launch costs. The problem was caused by a line of code that converted a 64-bit floating point number into a 16-bit signed integer. This caused an overflow which crashed the program and ultimately, the rocket.
3. In 1999, the Mars Climate Orbiter, which was launched by NASA to orbit Mars and study the climate, went on the wrong trajectory that brought it too near to Mars. It lost contact with mission control and was either destroyed in Mars' atmosphere or re-entered space at some unknown location. This was caused by the software engineers assuming all variables were in SI units. However, one group of engineers had used Imperial units instead. This caused a problem only when the calculations concerned with achieving orbit around Mars were carried out. The cost of this project was US$125 million.
4. In 2011, the UK government scrapped the National Programme for IT in the NHS (National Health System), which had been commissioned in 2002, because the project failed to produce a workable system. An estimated £12 billion had been spent on the project, whereas the initial estimate for the cost was £3 billion. This was not the fault of the software engineers, but the codes of ethics or codes of guidance also refer to project management as well, and this type of failure should not have occurred.

In the above examples, there was no public concern with the ethics of the project itself, only how it was carried out. However, there are areas associated with computer-based systems where there is public concern about the nature of the project or where it led, intentionally or not. Here are some examples to consider:

- Powerful commercial companies being able to exert pressure on less powerful companies to ensure that the powerful company's products are used even if alternatives are more suitable or less costly.
- Companies providing systems that do not guarantee security against unauthorised access.
- Organisations that try to conceal information about a security breach that has occurred in their systems.
- Private data transmitted by individuals to other individuals being stored and made available to security services.

- Social media sites allowing abusive or illegal content to be transmitted.
- Search engines providing search results with no concern about the quality of the content.

(No examples have been provided for these. You should be able to find examples on your own using a web search.)

Public attitude to such concerns varies with country and time. This makes it difficult for an individual software engineer to make a judgment with respect to public good. Even if the judgment is that a company is not acting in the public good, it will be difficult for one person to exert an influence in the company. There are examples where the life of such individuals, who have taken action, have been severely affected.

Legal frameworks

Most countries have laws covering the illegal usage of technology. In Singapore, some of these laws include the following:

- The Computer Misuse Act[4] covers, among other things, hacking, sabotaging computers, accessing and distributing confidential data, copyright infringement including installing and distributing pirated software, cyber-stalking, harassment or online grooming, and credit card fraud.
- The Cybersecurity Act[5] establishes a legal framework for oversight and maintenance of national cybersecurity, including establishing a cybersecurity regulator, imposing cybersecurity obligations on organisations providing critical and essential services, licensing and regulating cybersecurity service providers, and providing a framework for sharing cybersecurity information.
- The Personal Data Protection Act (PDPA)[6] covers the collection, use, disclosure and care of personal data.

Ownership and copyright

Copyright is a formal recognition of ownership. If a person creates and publishes some work that has some element of originality, that person becomes the owner and can claim copyright. (An exception is if the person works for an organisation. An organisation can claim copyright for published work created by one or more people working for the orgsanisation.)

Copyright can apply to any of:

- Literary (written) work
- Musical composition
- Film
- Music recording
- Radio or TV broadcase
- Works of art
- Computer programs

---

[4] https://sso.agc.gov.sg/Act/CMA1993

[5] https://sso.agc.gov.sg/Acts-Supp/9-2018/ and https://www.csa.gov.sg/legislation/cybersecurity-act

[6] https://sso.agc.gov.sg/Act/PDPA2012 and https://www.pdpc.gov.sg/Overview-of-PDPA/The-Legislation/Personal-Data-Protection-Act

Copyright cannot apply to an idea, and it cannot be claimed on any part of a published work that was previously published by a different person or organisation.

There are two reasons copyright law exists:

1. Creative work takes time and effort and requires original thinking, so the creator deserves the opportunity to earn money from it.
2. It is unfair for another person or organisation to reproduce the work and make money from it without paying the original creator.

Singapore's Copyright Act can be found online[7]. While copyright laws vary from country to country in details, there is an international agreement that copyright laws cannot be avoided, for example, by republishing someone else's work in a different country without the original copyright holder's permission. Typical copyright laws include:

- A requirement for registration recording the date of creation of the work
- A defined period when the copyright will apply (usually a fixed amount of time after the creation of the work, or the creator's death).
- A policy to be applied if the person holding copyright dies.
- An agreed method for indicating the copyright, for example, using the © symbol.

While the copyright is in place, there will be implications for how the work can be used. The copyright holder can include a statement concerning how the work might be used. For instance, Section 10.02 of the ACM/IEEE code of ethics describes how it should be reproduced.

Other conditions describe what is permissible when the work has not been sold. For example, if someone has bought a copy of a copyrighted product (such as a book), there is no restriction on making copies as long as they are only for that person's use. Other regulations relate to, for example, books in a library, where people can photocopy a limited amount of a book.

The consequences and development of the Internet and World Wide Web

Before the internet, copyright breaches tended to happen mainly in two ways.

1. People with a tape cassette recorder could record a radio broadcast or make copies of other people's cassette tapes and vinyl records.
2. People photocopied printed material such as books and articles.

In the early days of the Internet, illegal copying, or piracy, of movies, TV shows, and music was commonplace, when these media were originally intended to be sold to distributors (cinemas, TV and radio stations) or bought (as CDs or DVDs). The scale and ease of illegal distribution was much larger than it had previously been and began to affect the profitability of the creators.

The initial response was to create **Digital Rights Management (DRM)** to counter such activity. For instance, DRM made CDs playable on CD players but not on computers, to prevent ripping. This was done by encryption or deliberately including some damaged sectors. Unfortunately, they did not guarantee the prevention of piracy.

[7] https://sso.agc.gov.sg/Act/CA1987

The major mechanism for piracy of media content is the widespread use of peer-to-peer (P2P) sharing. As a result, ISPs have been asked to monitor P2P technology usage and report it to interested parties. This, however, has been argued as being a breach of privacy.

The current commercial model for many content producers is to make some content available online for free, on streaming sites such as Youtube, and allow buyers to pay for remaining content at an affordable rate.

There are many resources on the internet describing copyright law for laymen and how it applies to content produced on the internet. Here are two examples:

- https://www.youtube.com/playlist?list=PL8dPuuaLjXtMwV2btpcij8S3YohW9gUGN (Playlist on Intellectual Property Rights by CrashCourse)
- https://www.youtube.com/watch?v=1Jwo5qc78QU (Youtube's Copyright System Isn't Broken. The World's Is by Tom Scott)

Software licensing

Commercial software is no different from any other commercial product, in the sense that it is created and sold by a company to make a profit. However, there is one key difference. When you buy, for example, a phone, you own the phone. However, if you buy software, you do not own the software. Instead, what you are buying is a **license** that allows you to use the software. There are some different models of this, including:

- Paying a fee for each individual copy of the software
- An organisation might buy a site license which allows a fixed number of copies of the software to run on the organisation's computers at any one time
- Special rates might be offered for educational use

In some instances, the license may be provided free-of-charge. There are two possibilities.

- **Shareware** is commercial software which is made available on trial basis for a limited time. During this trial period, either the full version or a limited version might be available, or a beta test of a new version.
- **Freeware** does not have a time limit on the free usage. It could be the full software or an earlier version.

Whether the license is paid or free, there will be limitations on the use of the software and the user will not be provided with the source code.

Open or free licensing is carried out by two global Non-Profit Organisations with slightly different philosophies, but both of them are **open source**, meaning that users have access to the source code, and are free to use it, modify it, copy it, or distribute it in accordance with the terms.

The **Open Source Initiative** is a movement to make open source software available. The philosophy here is that using open source software will allow collaborative development of software to take place. The software is normally available free-of-charge.

The **Free Software Foundation** is named because the philosophy is that users should be free to use the software in any way they wish. The software may not be free – there is usually a small fee to cover distribution costs. One special feature of the license is something called **copyleft**,

which is the condition that if users modify the source code, the modified version must be made available for other users under the same conditions.

When should you use commercial software?	When should you use open source software?
• The software is available for immediate use and provides the functionality required. • The software has been created to be used in conjunction with already installed software. • There will be continuous maintenance and support provided. • Shareware might allow suggestions to be made as to how the software can be improved. • Freeware can often offer sufficient functionality to serve your needs	• The full functionality can be provide for at most a nominal cost. • The software could provide the required functionality with just a few modifications to the source code. • A consortium of developers are collaborating in producing a new software suite. • The future development of the software or the continuous provision of the existing software is controlled by the user.

Examples of software (fill in your own):

Commercial paid • Microsoft Office • • •	Open source or free software • Mozilla Firefox • Open Office • Python • MongoDB • • • •
Shareware/Freeware • Google Chrome • • •	

Impact of computing and technology

The impact of computing on society can be roughly divided into four overlapping areas: social, ethical, legal, and economic.

Area	**Examples of impact (fill in your own)**
Social	Internet and modern communication technology: Allowing people who are physically distant to communicate with one another easily and immediately Allows access to information to people far more easily and quickly than traditional media Allows collaboration between teams which are geographically separated Artificial intelligence

	Helps with data analysis, such as predicting where crimes are likely to take place, calculating shapes of proteins for medical purposes, optical character recognition to help visually handicapped people read text, etc.
Ethical	Programming self-driving vehicles to make decisions when an accident is imminent leads to an ethical dilemma, e.g. should the car prioritise the driver's life or the pedestrian's? Artificial intelligence tends to exaggerate human biases. Does this lead to biases in predictions?
Legal	Creation of laws pertaining to cybersecurity, data collection and protection Modification of copyright laws (see above section on copyright) Current laws unequipped to handle artificial intelligence?
Economic	Cost savings due to automation of repetitive tasks, but job losses for the same reason Creation of new technology-related jobs in existing companies

Suggested videos:

https://www.youtube.com/watch?v=ltCVp1ic-L8 (The rise of human-computer cooperation - Shyam Sankar)

https://www.youtube.com/watch?v=MnT1xgZgkpk (What happens when our computers get smarter than we are? | Nick Bostrom)

https://www.youtube.com/watch?v=t4kyRyKyOpo (The wonderful and terrifying implications of computers that can learn | Jeremy Howard)

https://www.youtube.com/watch?v=hQigUH0vZSE (A funny look at the unintended consequences of technology | Chuck Nice)

https://www.youtube.com/watch?v=5xflVUa4M_8 (Robert Reich: "Preparing Our Economy for the Impact of Automation & AI" | Talks at Google)

https://www.youtube.com/watch?v=oYmKOgeoOz4 (Max Tegmark: "Life 3.0: Being Human in the Age of AI" | Talks at Google)

https://www.youtube.com/watch?v=JcC5OV_oA1s (Amir Husain: "The Sentient Machine: The Coming Age of Artificial Intelligence" | Talks at Google)

https://www.youtube.com/watch?v=leX541Dr2rU (There is No Algorithm for Truth - with Tom Scott)

https://www.youtube.com/watch?v=Rzhpf1Ai7Z4 (Should Computers Run the World? - with Hannah Fry)

https://www.youtube.com/watch?v=TtisQ9yZ2zo (Christmas Lectures 2019: How to Bend the Rules - Hannah Fry)

2021 JC2 H2 Computing 9569
33. Network Security

Malware

Malware is a short form for malicious software. Examples include viruses, spyware, ransomware, worms and trojans.

Malware aims to damage computer systems and/or to gain unauthorised access to them. For instance, a computer user may unwittingly download a file containing a virus from the Internet and run it. Without anti-virus software to stop it, the virus will infect the computer and may cause the computer to crash or have its data deleted. If the computer happens to be an important server, there can be great damage to the company concerned.

A spyware is a hidden program that secretly collects information about its user and transmits information to attackers without the user's knowledge. Signs that your computer may be running spyware:

- It runs slower than usual
- The web browser automatically accesses an unfamiliar website regularly
- Anti-spyware software crashes or cannot work properly
- The camera turns on without any user input

Different kinds of malware (technical details are not in syllabus):

- Ransomware is a type of malicious software that threatens to publish the victim's data or perpetually block access to it unless a ransom is paid. While some simple ransomware may lock the system in a way that is not difficult for a knowledgeable person to reverse, more advanced malware uses a technique called cryptoviral extortion, which encrypts the victim's files, making them inaccessible, and demands a ransom payment to decrypt them.
- A virus attaches itself to a program or file so it can spread from one computer to another. Some only have mildly annoying effects, but others can cause severe damage to hardware, software, or files. Usually they are attached to executable files, so even though they exist on a computer, they will not infect the computer until the executable file is run or opened. They are spread by sharing infected files, through downloads, email attachments, file-sharing, etc.
- A worm may be considered to be a special class of virus. The main difference is that they are standalone software and can spread from one computer to another **without** action by a host program or a person. A worm uses file or data transport features on the system. It has the capability to replicate itself and thus send many copies of itself to many different computers (e.g. an entire address book). It then replicates itself in the recipients' computers and sends itself out in *their* address books, etc. As a result, the worm ends up consuming large amounts of system memory or network bandwidth, causing servers to be overwhelmed and stop responding. More recently, worms can also act as spyware, allowing malicious users to control a computer remotely.[1]
- A trojan (short for Trojan horse) is a destructive program that looks like a genuine application. It opens a backdoor entry to the computer which gives access to malicious users or programs, allowing them to extract confidential or personal information.

[1] The Blaster Worm spread in August 2003 among computers running Windows XP and Windows 2000 operating systems. It was programmed to create a DDOS attack against the Windows Update site. Microsoft temporarily shut down the site to minimize potential effects of the worm.

Denial of Service (DOS) attacks

A denial of service attack on a server happens when a hacker sends many requests to the server, causing it to overload, and thus preventing legitimate users from accessing it. This also causes the server to use up unnecessary resources and disrupts normal operations on the server.

A hacker may also make use of malware to control multiple computers to attack a server at the same time. This is called a distributed denial of service (DDOS) attack.[2]

Protection mechanisms: Firewalls

A **firewall** is a system that is designed to prevent unauthorized access to a private network by filtering the information that comes in from the internet.

A firewall blocks unwanted traffic and permits wanted traffic. Its purpose is to create a safety barrier between a private network and the public internet. There may be hackers and malicious traffic that may try to penetrate into a private network to cause harm. A firewall is the main component on a network to prevent this.

It is especially important to a large organization that has many computers and servers in them, because the company would not want all those devices accessible to everyone on the internet where a hacker can come in and disrupt that organization.

A firewall that's used in computer networks is very similar to how a firewall works in a building. A firewall in a building provides a barrier so that in the event of an actual fire, on either side of a building, the firewall is there to keep the fire contained and to keep it from spreading over to the other side, preventing the fire from destroying the entire building. A network firewall works in a similar way as a building firewall. It stops harmful activity before it can spread into the other side of the firewall and cause harm to a private network.

A firewall works by filtering the incoming network data and determines by its rules if it is allowed to enter a network. These rules are also known as an **access control list**. They are customizable and are determined by the network administrator. The administrator decides not only what can enter a network but also what can leave a network.

For example, an access control list may show a list of IP addresses that have been allowed or denied by this firewall, so that traffic from some IP addresses are allowed to enter this network but traffic from other IP addresses may been denied. Rules may also be based on domain names, protocols, programs, ports, and keywords.

There are different types of firewalls

- **Host-based firewall.** This is a software firewall. This is installed on a computer and it protects that computer only and nothing else. For example, later versions of Microsoft operating systems come pre-packaged with a host-baseball firewall. There are also

[2] On October 22 and 24, 2016, a DDOS attack brought down Starhub's broadband network. This was the first time a major attack took place on a Singaporean telco's infrastructure. Some Starhub's subscribers' machines were infected with malware that repeatedly sent requests to Starhub's DNS. Since they came from subscribers' machines, Starhub's server believed they were legitimate and did not block them. The server was eventually overwhelmed and went down.

third party host based firewalls can be purchased and installed on a computer. Many antivirus programs will have a built in host-based firewall.

- **Network-based firewall.** This is a combination of hardware and software, and it operates at the network layer. It is placed between a private network and the public internet Unlike a host-based firewall, where it only protects that computer, a network-based firewall protects the entire network, and it does this through management rules that are applied to the entire network so that any harmful activity can be stopped before it reaches the computers. Network-based firewalls can be a stand-alone product, an arrangement which is mainly used by large organizations. They can also be built-in as a component of a router, which is mainly used by smaller organizations. They could also be deployed in a service provider's cloud infrastructure.

Most organizations will use both network- based and host-based of firewalls. They will use a network-based firewall to protect the entire network, and they will also use host-based firewalls for their individual protection for their computers and servers. In the event that harmful data happens to get passed the network firewall, the host based firewalls on each computer can still stop it.

Protection Mechanisms: Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS)

The IDS is a passive system that scans incoming traffic. Once the IDS identifies dangerous or suspicious traffic it can send an alert but leaves the action to the IPS.

The IPS is able to actively block or prevent intrusions. Actions taken by IPS:

- **Inspection and investigation.** Inspection can include signature-based inspection and a statistical anomaly-based inspection. Investigation includes analyzing suspicious package and activities.

- **Action.** Once unwelcome packets are identified, the IPS would either put them in quarantine or simply drop them.

- **Logs and Reports.** Like many security devices IPS can log attacks and send reports.

IDS and IPS are not necessarily two separate physical devices. They can be combined into one device They can be also be combined with other devices such as firewalls or routers into a single device.

Encryption, Digital Signatures, and Authentication

Network Applications handle data. How do we ensure data is safely transmitted? We can use encryption, digital signature and authentication.

Encryption protects data by encoding[3] it such that a secret key is required to decode[4] the data. The decoding process is also known as decryption. Before decryption, encrypted data appears as random, meaningless data. This provides security for the computer system, as only the authorised users who have the secret key can access the data. However, encryption does not prevent the hacker from deleting the data from the computer system.

[3] Encoding is the process of converting the data or a given sequence of characters, symbols, letters etc, into a specified format.
[4] Decoding is the reverse of encoding.

By itself, encryption does not verify the sender of a message. One way to verify the sender is to use a private and public key system. In such a system, users publicly reveal their **public key**, which is unique to each user and is used to encrypt messages intended for them. Once a message is encrypted, it can be decrypted using only the user's **private key**, which is known only by the user. Encryption using a public key is a one-way process. No one, not even the sender, can decrypt the message without knowledge of the private key or the original message.

A **digital signature** works in a similar way. A data file is hashed and encrypted using the sender's private key to form a signature. The original (unhashed) data file and signature are sent to the recipient. The recipient hashes the received data file, and also decrypts the signature using the sender's public key. If the hashed data file and the decrypted signature correspond, the data and signature are valid.

This verifies that a message comes from the intended sender and has not been changed by a third-party.

Finally, a network application requires authentication to check the identity of the user requesting to enter a system, ensuring only those with valid credentials can access the system. Common ways of authentication include:

- passwords
- biometrics, for example: fingerprints, facial recognition, iris scans
- token values, such as from a physical device, a mobile phone or a software application

Some applications use two-factor authentication (2FA), which uses two different ways of authentication for better security.