**Python variables and functions:**

Python as a language requires less syntax to perform a lot of operations than C. For example, each line in python does not need to end with a semicolon and variable types don’t always need to be expressly declared such as if you assign a string piece of data to a new variable the variable will automatically declare itself as a string.

**Assignment** is done using an = for example:

player\_choice = “rock”

The convention for variable names is to use snake\_case

To write a String, you can use either single or double quotes such as ‘String’ and “String”.

Variables cannot start with a number. Variables can be composed of uppercase and lowercase letters, numbers and an \_. Variables can be assigned to different data types such as integers (full numbers), strings (text), floats (decimal numbers) and Booleans (true or false). Variable names cannot be predefined keywords such as if or print.

In python, indentation is very important. When defining a function, any code that is indented more than the declaration line is considered inside of the function which is used instead of the { and } used in C to show what code is inside of the function.

**Functions** and **procedures** are declared using the keyword def for example:

def get\_choices():

//code inside the function

//code outside the function

Where parameters can be defined inside the () and the : is used to show that following indented code will belong to this function (and the colon can also be used in things like if statements to show the following indented code will be part of this if statement).

The return keyword is used to return a value from a function for example:

def function():

value = 4

return value

When a return statement is reached, no further lines will be run inside of the function.

To call a subroutine, you write the name of the subroutine with brackets containing any arguments for example:

def the\_function(value):

value = value + 1

return value

the\_function(1)

The final line is where the function is called which will return the number 2.

The **print** function is used to display text to the console for example

print(“words”)

def word\_function():

return “words”

print(word\_function)

Both print statements will print the word “words” to the console.

Parameters are able to have default values if no arguments are inputted to a function like this:

def hello(name = “Stranger”):

print(“Hello ” + name)

Where

hello(“Oscar”) would print “Hello Oscar” and

hello() would print “Hello Stranger” as this is the default value.

Parameters are passed by reference usually however integers, Booleans, floats, strings and tuples are immutable meaning that their value is not changed if passed as an argument into a function e.g.

def addtwo(value):

value += 2

val = 1

addtwo(val)

print(val) This would still print 1.

This is because the value parameter is created as a separate variable to the val argument which was passed in however if this was done with a mutable data type like a list, this would update the original list.

You can use a return statement without including anything after it in order to end the function and return to the original line of code where the function was called without returning a value such as if a starting condition for the function was not met.

You can return multiple values by separating them with commas for example:

def subroutine(number)

number1 = number \* 10

number2 = number \* 100

return number, number1, number2

print(subroutine(1)) which would print (1, 10, 100) in a list.

**Variable scope:**

If a variable is declared outside of all indentations, then all code below it can access it including inside of subroutines. This is called a global variable.

If a variable is declared inside of a function, then that variable can only be accessed inside that function and inside of all more deeply indented code such as selection or a function call inside the function. This is called a local variable.

Variables always can only be accessed by code below its declaration no matter the scope.

**Nested functions:**

Functions can be defined inside of other functions which means that they can be called infinite times inside of that function, but they cannot be called outside of that function. This is useful to create a subroutine which is useful inside of a specific function but has no use outside of it. e.g.

def count\_to(end\_number):

def counting(the\_number):

print(the\_number)

the\_number = the\_number + 1

return the\_number

current\_number = 1

while(current\_number <= end\_number):

current\_number = counting(current\_number)

count\_to(10)

The count\_to function counts to a certain number starting from 1 and it uses the counting function defined inside to print the current number and increment it by 1.

To access a variable defined inside of an inner function from an outer function you must define that function as nonlocal e.g.

def function1():

variable = 0

def function2():

nonlocal variable

variable = variable + 1

print(variable)

function2()

function1()

Declaring “variable” as nonlocal means that although it was declared in the inner function it can still be set to 0 in the outer function.

**Closures:**

Closures are where you return a nested function from an inner function which means that the nested function has access to the variables defined in the inner function despite being returned outside of it for example:

def function1():

variable = 0

def function2():

nonlocal variable

variable = variable + 1

return variable

return function2()

function2\_variable = function1()

print(function2\_variable) #prints 1 then

print(function2\_variable) #prints 2 after as it is updating “variable” still.

**Dictionaries, inputs and random:**

**Dictionaries** store data in key value pairs. This is where keys can be listed alongside values which they are set equal to for example:

dict = {“name”: “beau”, “colour”: “blue”, “age”: 17}

The value in the pair can refer to a variable or other data structure and data can be retrieved from a dictionary such as in the example:

dict[“name”]

this will return the string “beau” as referring to the key returns the value it is tied to.

To edit the values inside a dictionary you can do this by referring to the keys for example:

dict[“name”] = “John”

This would change the value “beau” to instead be “John”.

You can use the get method to get a value from a key as well for example:

dict = {“name”: “beau”, “colour”: “blue”, “age”: 17}

dict.get(“name”) would return “beau”

A benefit of this function is you can set a second parameter to be a default value if no corresponding values are found in the dictionary for example:

dict = {“name”: “beau”, “colour”: “blue”, “age”: 17}

dict.get(“surname”, “Turnbull”)

This would return “Turnbull” as there is no surname key present inside the dictionary and “Turnbull” is set as the default value.

You can use the pop() method to get a value and delete the key corresponding to it for example:

dict = {“name”: “beau”, “colour”: “blue”, “age”: 17}

dict.pop(“name”) this will return beau and delete the name key entirely.

The popitem() method returns the last key-value pair inserted into the dictionary and removes it for example:

dict = {“name”: “beau”, “colour”: “blue”, “age”: 17}

dict.popitem() would return (“age”, 17) as this is the last item that was added to the dictionary and would leave the dictionary as {{“name”: “beau”, “colour”: “blue”}.

You can use the in operator to see if a key is contained within a dictionary. For example:

“colour” in dict would return True.

You can get a list of the keys in a dictionary using the .keys() method for example:

dict = {“name”: “beau”, “colour”: “blue”, “age”: 17}

dict.keys() would return “dict\_keys([“name”, “colour”, “age”]) and you can use the list() function to return just the list part.

You can use the .values() function to print all of the values in a dictionary as well.

You can use the .items() function to return all the items (keys and values) in the dictionary in the format of a list.

You can use the len() function to see how many key-value pairs are in the dictionary.

You can use insert a new key-value pair into a dictionary like so:

dict[“new thing”] = “the\_thing”

Which will insert the new key-value pair to the end of the dictionary.

You can delete a key-value pair using del e.g.

del dict[“colour”] which will remove the colour key from the list.

You can create a copy of the dictionary using the .copy() function for example:

dict\_copy = dict.copy()

The **input** function is used to get a string input from the user which they type for example:

player\_input = input(“Please input something here”)

where the parameter of the function is printed onto the screen and whatever the user inputs is assigned to the age variable.

Python has a large quantity of libraries which can be linked using the **import** keyword for example:

import random

which imports the “random” library. The random library can be used to generate random numbers or random indexes.

**Lists and selection:**

A **list** is used to store multiple items of data of any data type for example

the\_list = [“name1”, “name2”, “name3”]

To generate a random item from a list you would do:

random\_choice = random.choice(the\_list)

A list containing multiple data types would look like:

list = [12, 2.5, “Twenty-four”]

To see if a particular item is contained inside of a list you would use the in function for example:

list = [12, 2.5, “Twenty-four”]

print(“Twenty-four” in list) would print True.

Whereas

print(13 in list) would print False.

You can also use indexes with lists for example:

list[1] would return 2.5.

You can also use indexes to edit the items inside of a list. for example:

list[1] = 88.3 would change the 2.5 in the list into 88.3. A negative index begins from the back of the list and you can use a : to select a range of items inside the list.

You can also use the len() function to find out how many items are in the list. You can use the append function to add an item to the end of a list as a new index. You can use the extend() function to add multiple items to a list by passing in the items as a list as a parameter to be collectively appended to the end. You can use += to add a list to another list as well which is the same as the extend function. For example:

list += [“item1”, “item2”]

and then the list would contain [12, 2.5, “Twenty-four”,“item1”,“item2”]

and list.extend([“item1”, “item2”]) would do the same thing.

The .remove() method can be used to remove an item from a list, for example:

list.remove(“item1”)

Would leave the list as [12, 2.5, “Twenty-four”,“item2”]

The .pop() function removes and returns a single item from a list. For example:

list.pop()

Would return “item2” and leave the list as [12, 2.5, “Twenty-four”]

You can use the insert function to add an item into the middle of a list. For example:

list = [1,2,4,5]

list.insert(2,5)

Would insert the number 5 into the second index of the list and move all of the rightmost items one index to the right including the item in index 2 which would turn the list into [1,2,3,4,5].

To insert multiple items into the middle of a list you would do:

list = [1,4,5,6]

list[1:1] = [2,3]

List[1:1] starts at the first index but finishes at the first index so it does not replace any items already inside the list instead it just inserts the extra list inside the full list at index 1.

The sort() function can be used to sort a list. The sort() function only works when all items in the list are of the same data type e.g. it cannot sort strings and integers. An example of this would be:

list = [1,6,2,3,0]

list.sort()

This would change the list to [0,1,2,3,6].

The sort() function orders strings in alphabetical order and it counts uppercase letters as before lowercase letters e.g. beau would be placed after Beau as the capital B comes before lowercase b.

You can use list.sort(key = str.lower) to make sure all items are sorted ignoring cases.

To copy a list you can do:

list\_copy = list[:]

Where list\_copy is the same as list now.

To sort a list without modifying the original list just returning the sorted version you can use the sorted() function for example:

sorted(list, key=str.lower)

would return the sorted list where the first parameter is the list and the second parameter is the key.

**List compression** is used to create a new list from a previous one, possibly applying an operation to each one and also possibly only selecting a smaller number of items from the whole list e.g.

numbers = [1,2,3,4,5]

numbers\_power\_2 = [n\*\*2 for n in numbers]

Which would return [1,4,9,16,25] and the n\*\*2 is applied to each item in the list.

**If statements** are a form of selection which checks a given condition, and if the condition is true then it will run a segment of code and if it is false then it will not run the segment. Else statements can also be used to run a section of code if previous if statements are false. Elif statements are used to combine else if together which will run the segment of code if the previous if statements are false and if the new given statement is met. For example:

number1 = 4

number2 = 5

if number1 > number2:

print(number1 + “ > ” + number2)

elif number1 < number2:

print(number1 + “ < ” + number2)

else:

print(number1 + “ = ” + number2)

which will print “4 < 5” as 5 is greater than 4 which is the code inside of the elif.

It is possible to use **nested if statements** to run a section of code only if a condition is met and then if a second condition is also met. For example:

if player == “rock”:

if computer == “paper”:

print(“You lose”)

elif computer == “scissors”:

print(“You win!”)

This means that this entire section of code will only run if the player variable is equal to rock and inside of that, You lose will be printed if computer equals paper whereas you win will be printed if computer equals scissors.

Indentation can be done with any number of spaces as long as it is consistent with the rest of the program.

**String handling:**

The + sign can be used to concatenate two strings together which means to combine them into a single string. For example:

new\_string = “This string” + “ That string”

This will make new\_string == “This string That string”

The += sign can be used to add to the end of a string . For example:

name = “Joe”

name += “Smith”

now the name will equal “Joe Smith”

f-strings in python combines text and variables in a single string. For example:

age = 25

print(f“Jim is {age} years old.”)

Where any variables are contained in the {} and there must be a f at the start of the f-strings.

If you use 3 quotation marks in your string e.g. “““String”””

it will keep the new lines in it e.g.

“““This is a

long

string”””

will print as:

This is a

long

string

You can also use \n to add a new line.

There are methods in python to deal with strings.

.upper() can be used to make a string uppercase for example:

“cat”.upper() will return CAT.

.lower() can be used to make a string lowercase for example:

“CaT1” will return cat1.

.title() can be used to make the first letter of each word uppercase for example:

“this is a STRING”.title() will return This Is A String.

You can also use .isupper(), .islower() or .istitle() to check if a given string fits one of those requirements.

Additionally,

.isalpha() checks if a string contains only characters and is not empty.

.isalnum() checks if a string contains characters or digits and is not empty.

.isdecimal() checks if a string contains only numbers and is not empty.

.startsswith() checks if a string starts with a specified substring.

.endswith() checks if a string ends with a specified substring.

.replace() replaces part of a string.

.split() splits a string into two substrings divided by a specified character separator.

.strip() trims whitespace from a string.

.join() appends new letters to a string.

.find() finds the position of a substring inside of a string.

For example:

“THIS IS UPPERCASE”.isupper() will return true as it is uppercase however

“This is NOT a title”.istitle() will return false as it is not a title.

len() returns the length of a string and also works with a number of other data structures.

you can use in to see if a substring is contained inside of another string for example:

“super” in “batman” will return false as there is no “super” in “batman”.

Some things like quotation marks cannot directly be written into strings as python will read that as you trying to end the string so to put things like quotation marks into a string you use a \ for example:

“\“Is this a dagger I see before me, The handle towards my hand?\”, cried Macbeth.” would display:

“Is this a dagger I see before me, the handle towards my hand?”, cried Macbeth.

To include a \ in a string, you would write \\.

You can use [] with strings to handle individual indexes. For example:

the\_string = “Woah”

print(the\_string[1])

will print the letter o as this is the 1st letter in the string Woah as computers start counting from 0.

print(the\_string[-1]) will print h as this is the first letter starting from the end of the string.

print(the\_string[1:3]) will print a substring starting from index 1 and ending with index 3 however the final index is not included so this will print oa.

Including a blank space before the colon will make the substring start from the very beginning and including a blank space after will make the substring end at the very end.

**Comments, data types and operators:**

In python, comments are indicated using a # for example:

#This is a comment

print(“This is not a comment”) #However, this is a comment at the end of a line

Where comments are sections of code which are not run but instead used to provide additional information to the programmer about the code.

The **type** function returns the data type of a value. For example:

print(type(“A sentence”))

will print str which stands for String.

The **isinstance** function takes two parameters, a piece of data and the name of a data type and it will return true if the piece of data is an instance of that data type and it will return false if it is not.

You can **cast** pieces of data into different data types using type constructors for example:

float(4) will turn the int 4 into a float,

str(float(4.01)) will turn 4.01 into a float and then into a string and

int(“4”) will turn the string “4” into an integer.

The common data types in python are:

complex for complex numbers

bool for Booleans

list for lists

tuple for tuples

range for ranges

dict for dictionaries

set for sets

The common operators in python are:

= for assignment

is returns true if both objects are the same

in is used to show if a value is contained within a list or sequence known as the membership operator.

**Arithmetic operators** in python:

+ to add two numbers together

- to subtract a number from another number

\* to multiply two numbers together

/ to divide two numbers resulting in a float

% to divide two numbers returning only the remainder (modulo)

// to divide two numbers returning only the integer part (DIV or FLOOR)

\*\* to get one number to the power of another number (exponent)

- can be used to make a number negative as well

+= adds a number to a variable for example

age = 10

age += 8

age = age + 8

The last two lines here do the same process

The **comparison operators** that can be used in python are:

== which is true if both items are the same.

!= which is true if both items are NOT the same.

> which is true if the left item is greater than the right item.

< which is true if the right item is greater than the left item.

>= which is true if the left item is greater than or equal to the right item.

<= which is true if the right item is greater than or equal to the left item.

The **Boolean operators** that can be used in python are:

**and** which is true if both operands (statements) are true.

AND returns the second operand if the first operand is true or else it returns the first operand for example:

1 and 1 returns 1

0 and 1 returns 0

False and “hey” returns False

“bonjour” and “hello” returns “hello”

**or** which is true if at least one of the operands are true.

OR returns the first which is not false for example:

0 or 1 returns 1,

False or “random text” returns “random text” and

“hi” or “hey” returns “hi”

**not** which is true if the single operand is false (and false if the single statement is true).

The **bitwise operators** that can be used in python are:

& performs binary AND

| performs binary OR

^ performs binary XOR

~ performs binary NOT

<< performs a left shift

>> performs a right shift

The **ternary operator** in python is used for a simple conditional operation. For example it can replace the following code:

def is\_adult(age):

if age > 18:

return True

else:

return False

With this code instead:

def is\_adult(age):

return True if age > 18 else False

Which is much faster to write.

**Booleans and complex numbers:**

The bool data type can be either True or False.

Booleans can be used in selection as in an if statement, if the condition is true then a section of code will run but if the condition is false it will not for example:

done = True

if done:

print(“yes”)

else:

print(“no”)

this will print yes as the condition is true however if done was set to false then it would print no.

For Booleans True is true and False is false.

For integers, 0 is false and any other number is true including negatives.

For Strings “” (an empty string) is false and a non-empty string is true.

lists, tuples, sets and dictionaries are all false when empty.

You can use the type() function to see if a data type is a Boolean.

The any() function on a list returns true if any of the items inside are true.

The all() function returns true if all of the items inside the list are true.

**Complex numbers** are numbers with a real part (just a normal number) and an imaginary part (some multiple of √-1). For example, 2 + 3i where 2 is the real part and 3i is the imaginary part meaning 2 times √-1.

To create a complex number in python, you use the letter j to represent √-1. For example:

complex\_number = 3 + 2j

This will make a complex number with 3 as the real part and 2 as the imaginary part or you can do:

complex\_number = complex(3,2)

Which will make the same exact complex number.

You can use the .real() function to return the real part of a complex number and you can use the .imag() function to return the imaginary part of the complex number.

**Functions for numbers:**

The abs() function returns the absolute value of a number which is how far away from 0 it is. This means that it will make a negative number positive, and it will keep a positive number also positive.

The round() function returns the number rounded to the nearest integer if you only include a single argument. If you include 2 parameters, the first parameter will be the number you want to round whereas the second number will be the place which you want to round the number to where 0 is a whole number, 1 rounds the number to the nearest 10th and -1 rounds the number to the nearest 10.

Enums are a type of object which can be used to store multiple CONSTANT values and enums need to be imported from the enum library like so:

from enum import Enum

To construct an enum, you would do this:

class State(Enum):

INACTIVE = 0

ACTIVE = 1

The word state is the name of the enum.

What this means is that State.INACTIVE.value would equal 0 and State.ACTIVE.value = 1 using .value to get the value.

If you want to find which variable inside of the enum corresponds to a particular value, you would do State(1) which would return State.Active.

If you do State[‘ACTIVE’] it would also return State.ACTIVE and if you do State[‘ACTIVE’].value it would return 1. Enums are the only way to enforce constants in python as the values cannot be changed after initialisation whereas normal variables can always be overwritten.

If you do list(State) it will list all the possible values which are stored inside State and which constants correspond to those values and you can use len() to find out how many constants are stored inside the enum.

**Tuples and sets:**

**Tuples** are a data structure which allows you to create a list of immutable object which means that the tuple cannot be updated in any way once they are created. Tuples use () instead of [] for example:

tuple = (“item1”, “item2”)

This creates a tuple with the strings item1 and item2 inside it.

You can access elements inside of a tuple using indexes for example:

tuple[0] will return “item1” and

tuple.index(“item2”) will return 1.

You can also use the len() function with tuples, the in keyword and slices for example:

len(tuple) will return 2 and

“item1” in tuple will return True and

tuple[:1] will return “item1”.

The sorted function works with tuples as this does not modify the original list however the sort function does not work as tuples cannot be modified. You can use the + operator to create a new tuple using existing ones

**Sets** are a data structure like a list however it is unordered (elements have no index) and it does not allow duplicates. You can define a set like so:

names1 = {“Oscar”, “Toby”, “Jeff”}

You can find which items are shared inside of multiple sets using & (AND) e.g.

names1 = {“Oscar”, “Toby”, “Jeff”}

names2 = {“Oscar”}

print(names1 & names2) Which would print {“Oscar”}

You can find which items are inside either of the sets using | (OR or union) e.g.

names1 = {“Oscar”, “Toby”, “Jeff”}

names2 = {“Oscar”, “Bob”}

print(names1 | names2) Which would print {“Oscar”, “Toby”, “Jeff”, “Bob”}

You can find which items are NOT inside both of the sets but are in either one of them by using – e.g.

names1 = {“Oscar”, “Toby”, “Jeff”}

names2 = {“Oscar”}

print(names1 - names2) Which would print {“Toby”, “Jeff”}

You can check if a set is a subset or superset of another using > or < e.g.

names1 = {“Oscar”, “Toby”, “Jeff”}

names2 = {“Oscar”}

print(names1 > names2) Which would print True as names2 is a subset of names1 as all of the elements in names2 are contained in names1 however names1 also has additional items. names1 < names2 would print False.

You can use len() to count the amount of items in a set.

You can get a list of all of the items in a set by using the list() constructor e.g.

list(names1) would return (“Oscar”, “Toby”, “Jeff).

You can also use in to see if an item is contained within a set. As sets cannot contain duplicate values, the set

set1 = {“item1”, “item2”, “item1”} would become {“item1”, “item2”}

If you want to remove duplicate items from a list, you can convert it into a set and convert it back into a list.

**OOP in Python:**

**Objects:**

Almost everything in python is an object such as strings, lists, tuples, etc. Objects have attributes which are like variables to store data and methods which are like subroutines to perform operations. Methods and attributes can be accessed using a . after the object name.

For example, the int object has the .real() and .imag() methods to return the real and imaginary parts of an integer. The .bit\_length() method returns the amount of bits necessary to store the integer in binary.

The id() function allows you to inspect the location in memory of a particular object which will return an integer corresponding to its memory location.

Mutable objects in python are able to be changed and updated whereas immutable objects cannot. For example, a list in python can have its contents changed however an integer cannot and when you update an integer such as by adding a number to it, then this actually creates a brand new integer object to replace the previous integer.

**Classes:**

You can declare your own classes in python using the class keyword. For example:

class Dog:

def bark(self):

print(“woof”)

Where the entire class is defined inside the class keyword and the bark method is a method inside the dog class. The self parameter refers to the current object which is running the method so using self as a parameter means that the object which the method is called in will use its attributes inside of the method instead of running that method in another object.

To instantiate an object, you assign its constructor to a variable e.g.

doggy = Dog()

Where a new instance of the Dog class is instantiated as “doggy”.

You can use the type() function to return the type of an object so type(doggy) would return <class ‘\_\_main\_\_.Dog’> which shows that doggy is instatiated from the Dog class.

To create a constructor in python you use the \_\_init\_\_ keyword e.g.

class Dog:

def \_\_init\_\_(self, age):

self.age = age

You pass in the self argument and then any additional arguments which you wish to be used in the instantiation of the object where in the example above, the age attribute is set to a passed in age parameter.

To access an attribute contained inside an object, you can use a . e.g.

class Dog:

def \_\_init\_\_(self, name,age):

self.age = age

self.name = name

def roll\_over(self):

print(name + “ rolls over”)

doggy = Dog(“Bruno”,10)

print(doggy.age) which will print 10.

Also, you can call the roll\_over() method like so:

doggy.roll\_over() which will print “Bruno rolls over” as Bruno is passed in as the name attribute.

**Inheritance** can be used to give a new class all of the methods and attributes of an existing class. For example:

class Sheep:

def baa(self):

print(“baa baa baa”)

def get\_colour(self):

return colour

class Black\_sheep(Sheep):  
 def \_\_init\_\_(self):

self.colour = “black”

class Grey\_sheep(Sheep):  
 def \_\_init\_\_(self):

self.colour = “grey”

class white\_sheep(Sheep):  
 def \_\_init\_\_(self):

self.colour = “white”

Inheritance is used by using () in the initialisation of a class where the class inside the brackets is the one the child class inherits from. In the example, there is a parent Sheep class which the other classes inherit from which means all the other classes have the baa() method to print the baa text and also a get\_colour() method to return the colour of the sheep and then each child class is instantiated with a different colour of wool meaning that each one will return a different string when the get\_colour() method is run but they will all print the same “baa baa baa” text.

**Polymorphism:**

The same method can be defined for multiple classes and do different things depending on what class it is part of. This could be useful if you have a list of objects and you want to use the same method with all of them and it does not matter which class they belong to if they share a method with the same name e.g.

list = [object1, object2, object3]

for object in list:

object.the\_method

This will do the method “the\_method” for each object in the list no matter which class any of the objects belong to.

**Operator overloading** can be used to compare attributes in different objects. e.g.

def \_\_gt\_\_(self,other):

return True if self.value > other.value else False

which will take in another object as a parameter and will compare the attribute called value inside of them BUT \_\_gt\_\_ is a special Name which refers to the greater than symbol which means calling the method can be done with > instead of a typical method call e.g.

object1 > object2 will actually call the method on both of them instead of a normal > evaluation. You can do this with other symbols such as adding, subtracting, mod, right shift, etc.

**Iteration:**

Loops are ways to repeat a section of code a number of times either using **while** loops or **for** loops. A while loop continuously repeats a section of code until its condition is False. For example:

condition = True

while condition == True:

print(“Looping forever”)

This while loop will repeat forever as the condition will always remain True however the following code:

condition = True

while condition == True:

print(“Looping once”)

condition = False

Will only loop once as the condition is immediately set to false when it is first run.

For loops repeat a certain amount of times which is predetermined such as running 5 times or repeating for each item inside of a list. For example:

items = [1, 2, 3, 4]

for item in items:

print(item)

This will repeat once for each item inside of the list “items” and each item will be printed when it is reached so this will print:

1

2

3

4

To iterate a certain amount of times, you use the range() function. For example:

for item in range(15)

print(item)

This will start from 0 and repeat up to and not including the number 15 as follows:

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

The range() function returns a list starting from 0 just up to and not including the argument so range(5) would return:

[0,1,2,3,4]

You can use the enumerate() function to receive the index of the item in a list as well as the item itself when you are looping through a list for example:

items = [1, 2, 3, 4]

for index, item in enumerate(items):

print(index, item)

This will print:

0 1

1 2

2 3

3 4

as the index variable holds the index of the item and the item variable holds the item itself.

The continue keyword during iteration stops the current iteration and continues to the next iteration. T he break keyword stops the iteration entirely and moves onto the next instruction after it. For example:

for i in range(5)

if i == 2:

continue

print(item)

This will print the numbers 0 to 4 but when it reaches 2 it will not print it and continue. An example of the break keyword is:

for i in range(5)

if i == 2:

break

print(item)

This will print the numbers 0 to 4 but when it reaches 2 it will stop the iteration completely meaning it will only print 0 and 1.

**Modules:**

Modules are python files. You can import a module from different files meaning that multiple python files can work together in one program which promotes code reuse and modularity. Typically, one python file is the main entry point and other files are modules which are used inside of it. You can import modules using the import keyword for example if you had two python files in the same folder called:

main.py

library.py

In the main file, you can import all of the functions from library.py by using:

import library

And then you can use all classes, data and subroutines from library.py.

If you have a subroutine called hello inside of library and you import it, to call this subroutine you would do:

import library

library.hello()

But if you just want to import the hello subroutine then you can do:

from library import hello

This would just import the hello subroutine and means that you can call it normally without specifying it is from library for example:

hello()

If library.py is in a subfolder inside the folder main.py is in, for main.py to access the library.py file, a file called “\_\_init.py\_\_” must be created inside of the subfolder so that python can access the modules inside of it and to import library.py if the subfolder is called “Subfolder” you would do:

from Subfolder import library

Which would import the library.

To just get the hello subroutine, you would do:

from Subfolder.library import hello

The standard libraries in python which can be used contain:

math for maths functionality

re for regular expressions

json to work with JSON

datetime to work with dates and time format

sqlite3 to use SQlite

os for operating system utilities

random for random number generation

statistics for statistical functionality

requests to perform HTTP network requests

http to create HTTP servers

urllib to manage urls

For example to use the math library you can do:

import math

math.sqrt(9)

and the sqrt function square roots a number leaving the result as 3.

You can also do from math import sqrt to just import the sqrt function.

**Accepting command line arguments:**

To run a python file in command line, you first write python and then the name of the python file so if you wanted to run a file called main.py you would write:

python main.py

To assist with running python code in the command line interface, inside of the python file you can import the sys module. You can use the argv keyword to return a list of all arguments that were passed into the program e.g.

print(sys.argv) which will print a list of all typed arguments which can be used to make the program respond to different user commands which are not inbuilt python ones in command line. For example:

import sys

print(“Please input your name”)

name = sys.argv(1)

print(“Hello” + name)

This will take the first thing that the user inputs other than the name of the python file and set it equal to the name variable and then that is printed back to the user.

An easier way to get inputs from the user using command line interface is by importing the argparse library in the python code. For example:

import argparse

parser = argparse.ArgumentParser( description = ‘this program prints the names of my dogs’)

# This line creates a new ArgumentParser object from the argparse library called parser and gives it a description.

parser.add\_argument( ‘-c’, ‘—color’, metavar= ‘color’, required=True, help = ‘the colour to search for’)

#This line tries to get an argument from the user when they type -c with the attribute named color and this is required.

args = parser.parse\_args()

#This parses the information taken from the user from the parser into the variable called args

print(args.color)

#this prints the color attribute stored inside the args object

Inside the command line interface, if the user inputs:

python main.py -c then they are able to input a colour and whatever they input is printed back. Because the color argument is set to required, if the user chooses to not input a colour or write -c then they will be met with an error message.

If inside the add argument line, you add a choices parameter then the user input is only able to be an item contained in the choices set e.g.

parser.add\_argument( ‘-c’, ‘—color’, metavar= ‘color’, required=True, choices={‘red’, ‘yellow’}, help = ‘the colour to search for’)

Then the input can only be one of red or yellow.

**Lambda functions:**

Lambda functions are functions which have no name and only have one expression as their body for example:

lambda num : num \* 2

Where “num” is the argument and num \* 2 is the returned expression.

Lambda functions have to return a value. Lambda functions can accept multiple arguments as well e.g.

lambda a, b : a \* b

Where a and b are the different arguments and a \* b is what is returned.

You cannot call lambda functions directly so they must be assigned to a variable to be called e.g.

multiply = lambda a,b : a \* b

and to call it you would do:

multiply(argument1, argument2)

Lambda functions are useful when combined with Map Filter and Reduce.

**Map, Filter and Reduce:**

**map()** is used to run a function on each item inside of a data structure such as a list and create a new list but the items in the new list may have been changed by the function being run on them e.g.

numbers = [1,2,3]

def double(a):

return a \* 2

result = map(double, numbers)

print(list(result))

Which will print[2,4,6] as the double function was used on each item in the list.

Lambda functions are commonly used for 1 line operations like this e.g.

numbers = [1,2,3]

double = lamda a : a \* 2

result = map(double, numbers)

print(list(result))

This would have the same result but is more simplified. To simplify this even more you can directly insert the lambda function inside of the map function as a parameter however this means it cannot be reused e.g.

numbers = [1,2,3]

result = map(lambda a: a \* 2, numbers)

print(list(result))

The list() function is used to convert the map object (which is an iterable) into a list so it can be printed.

**filter()** is used to take an iterable object and return another iterable object but filtering out some of the original items. For example:

numbers = [1,2,3]

def isEven(n)

return n % 2 == 0 (will return True or False)

result = filter(isEven, numbers)

print(list(result))

The filter function only keeps items in the original list which return True when passed into the filtering function, in this case “isEven()”.

The result of the example will be [2]

You can also use lambda functions instead for the filtering function.

**reduce()** is used to calculate a value out of a sequence such as a list. For example:

expenses = [ (‘Dinner’, 80), (‘Car repair’, 120) ]

sum = 0

for expense in expenses:

sum += expense[1]

print(sum)

This program, without using reduce(), calculates the sum of all the expenses which will be 200. Now, using reduce(), firstly you must import reduce from the functool library. Then, you pass in a lambda function and then the data structure you are using reduce on.

from functools import reduce

expenses = [ (‘Dinner’, 80), (‘Car repair’, 120) ]

sum = reduce(lambda a,b: a[1] + b[1], expenses)

print(sum)

The sum = line takes in two arguments where the first value is the accumulated value and the right value is the update value from the iterable where you reduce all of the numbers in the iterable into a single number by adding them together and the accumulated value starts as 0 and the update value is the number stored inside the tuple which is added each time.

**Recursion:**

Recursion is where a function calls itself. e.g. A factorial is where you multiply each number between a specified number and 1 together so 3! = 3 \* 2 \* 1 = 6. To do this using recursion you can do:

def factorial(n):

if n == 1:

return 1

return n \* factorial(n-1)

This means that factorial(3) will return 3 \* factorial(2) and factorial(2) returns 2 \* factorial(1) and factorial(1) returns 1 meaning that factorial(3) will eventually return 3 \* 2 \* 1 which is 6.

Recursive functions have to have a base case which is a case that the function converges to and when the function reaches the base case it will stop the recursion and unwind back to the original function call.

**Decorators:**

Decorators in python are a way to change how a function works and they themselves are a function which takes another function as a parameter e.g.

def logtime(func):

def wrapper():

print(“before”)

val = func()

print(“after”)

return val

return wrapper

@logtime

def hello():

print(hello)

The @ symbol is used to introduce a decorator and when the hello function is called, the decorator is also called. The decorator wraps the function in an inner function and returns the inner function. In the example above, The logtime function includes a section of code to be run before the hello function (here it prints “before”) and it also includes a section of code to be run after (here it prints “after”).

You would use a decorator when you want to change the functionality of a function without modifying the function itself such as for testing purposes like logging test performance and caching verification permissions or you can use them when you want to perform the same code on multiple different functions such as for validation.

**Docstrings:**

Docstrings are like comments and there are used to document and describe what sections of code should do. e.g.

class Dog:

“““A class representing a dog”””

def \_\_init\_\_(self, name, age):

“““initialise a new dog”””

self.name = name

self.age = age

The purpose of docstrings are to document the code and describe what it generally does whereas comments are used to explain what each line of code does.

It is common to put a docstring at the top of a file to explain what the file does and docstrings are able to go over multiple lines. You can use the help() function to receive the documentation for a certain class, function, etc e.g.

print(help(Dog)) which will provide lots of information about the dog class including its methods and attributes but this will also provide the docstrings contained inside the Dog class.

**Annotations:**

Annotations allow you to optionally define the expected data types for different variables e.g.

def increment(n: int) → int:

return n + 1

count: int = 0

This specifies that n should be an integer and the increment function should return an integer and the count variable should be an integer. Python ignores annotations but they are great to document what data type variables should be and a tool called mypy can be used to check for type errors and type mismatch errors with annotations.

**Exceptions:**

Exceptions can be used to attempt to run a section of code but if there is an error then it will run a different section of code instead using the keywords try and except. For example:

try:

#attempted section of code

except <ERROR1>:

#handler of ERROR1

except <ERROR2>:

#handler of ERROR2

except:

#handler of any other errors

else:

#this runs code after there are NO errors after the try section.

finally:

#This will always run in all cases if there is an error or not.

Instead of ERROR1 and ERROR2 you would write actual errors such as “EOFError” or “ZeroDivisionError”.

You can raise your own exception in your code by writing:

raise Exception(‘Oh no an error‼‼‼’)

Where the string inside the brackets will be the error message. For example:

number = int(input(“Enter a number which isn’t 4”))

if number != 4:

#run some code

else:

raise Exception(‘You inputted 4 which is not allowed’)

To simply print an error message without stopping the program you can do:

try:

raise Exception(‘An error’)

except Exception as error:

print(error)

You can also define a new class which inherits from the exception class e.g.

class pythonNotWorking(Exception):

pass #pass means there will be no code inside a function or class

try:

raise pythonNotWorking()

except pythonNotWorking:

print(‘Python isn’t working’)

To introduce your own errors. You can also add code inside the newly defined class to run that code when the exception is encountered e.g.

class pythonNotWorking(Exception):

print(“Oh no an error”)

pass

Which means Oh no an error will be printed when the pythonNotWorking exception is reached.

When opening a file you can do it like so:

filename = ‘/Users/Oscar/file.txt’

try:

file=open(filename,‘r’)

content = file.read()

print(content)

finally:

file.close()

Which means the file will always be closed even if there is an exception. However, you can use the **with** keyword like so:

filename = ‘/Users/Oscar/file.txt’

with open(filename, ‘r’) as file:

content = file.read()

print(content)

With makes sure to automatically close the file at the end and has implicit exception handling.

**Third party packages:**

pip can be used to install third party packages which can include things like libraries created by third party people and companies not affiliated with python to be used in code.

There are >270000 packages available using pip. pip can be used in command prompt terminal and the install command is used to install third-party packages e.g.

pip install requests

Typed into the terminal will install the “requests” package which will be available to use in all python files.

pip install -U (the package) will update the package to the latest version.

pip uninstall (the package) will uninstall a package.

pip show (the package) will provide lots of information about the package.