Decision Making and Functions

If Else

```
a = 33
b = 200
if b > a:
        print("b is greater than a")
elif:
        print(“whatever ”)
else:
        print(“I told u already”)
```

#python relies on indentation

#elif means if the previous conditions were not true, then try this condition.

#else keyword catches anything which isn't caught by the preceding conditions

We can use ‘and’ logical keyword

```
if a > b and c > a:
    print("Both conditions are True")
```

‘or’ keyword

```
if a > b or a > c:
    print("At least one of the conditions are True")
```

Python Loops

Python has two primitive loop commands:

- while loops
- for loops

while loop we can execute a set of statements as long as a condition is true.

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

Output:

1
2
3
4
5

The break Statement

With the break statement we can stop the loop even if the while condition is true:

```
i = 1
while i < 6:
  print(i)
  if (i == 3):
    break                    #exit loop when i=3
  i += 1
```

Output:

1
2
3

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

e.g. Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Output:

```
1
2
4
5
6
```

 # Note that number 3 is missing in the result

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

Less like the for keyword in other programming language, and works more like an iterator method

e.g.Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Output:
apple
banana
cherry

Even strings are iterable objects, they contain a sequence of characters:

e.g.

```
for x in "banana":
  print(x)
```

Guess the output here??

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
----------------------------------------
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The range() Function

- To loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

e.g.

```
for x in range(6):
    print(x)
```

will print **0 to 5** (not till 6)

Other options with range

```
for x in range(2, 6):
    print(x)
```

will print **2 to 5**

it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**)

```
for x in range(2, 30, 3):
    print(x)
```

Will print values 2 5 8 11 14 17 20 23 26 29

Else in For Loop:

The else keyword in a for loop specifies a block of code to be executed when the loop is finished: for e.g.

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Nested For Loops:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

Output:

```
red   apple
red   banana
red   cherry
big   apple
big   banana
big   cherry
tasty  apple
tasty  banana
tasty  cherry
```

Functions

- Block of organized, reusable code that is used to perform a single, related action
- Provide better modularity for your application
- High degree of code reusing

Defining a Function

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.

Syntax:

```
def functionname( parameters ):
   "function_docstring"
    function_suite
    return [expression]
```

Note:
Parameters have a positional behavior and you need to inform them in the same order that they were defined.

function1.py

Possibilities with a Function

- Assign a function to a variable. func_as_variable.py
- Define one function inside another function.func_in_func.py
- Pass a function as a parameter to another function.func_as_param.py
- A function can return another function.func_another_func.py

Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference.

It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

function_2_parameter.py

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

function_3_parameter.py

The parameter **mylist** is local to the function changeme. Changing mylist within the function does not affect mylist.

Function Arguments(Formal & Actual)

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

- Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.
- To call the function **printme()**, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

required_argument.py

Keyword Arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.
- keyword_argument.py

Default Arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed-

default_argument.py

Variable-length Arguments

- You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

- This tuple remains empty if no additional arguments are specified during the function call.
- variable_length_argument.py

Lambda

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

-Syntax

lambda *arguments* : *expression*

The expression is executed and the result is returned:

-Lambda functions can take any number of arguments:

e.g.

```
x = lambda a, b, c: a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

- The power of lambda is better shown when you use them as an anonymous function inside another function.
- Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:
- lambda1.py

- lambda2.py

Use lambda functions when an anonymous function is required for a short period of time.

Return Values

- To let a function return a value, use the return statement:

```
def my_function(x):
    return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Nested Functions

- nested_function.py
- nested_function2.py

Try Except

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the try statement.

try_except.py