Basics of Python

By the end of this practical, you will be able to

- Create and execute basic Python programs
- Comprehend and explain basic Python language features
- Access and modify files

Basic language features

Variables

A name that is used to denote something or a value is called a variable.

- Each variable has a name and a scope
- A variable will be declared upon assignment if it has not been declared before.
- A variable's name must start with 'a' to 'z', 'A' to 'Z' or '_', followed by any number of alpha-numeric characters or '_'
- A variable cannot be named using special keywords such as print, if, switch and etc.
- here are some valid variable names

```
x1
first_name
_one
```

- here are some invalid variable names

```
1x
first.name
print
```

Statements

A statement is a program instruction to be executed (most of the time).

- basic building blocks in python program (for most of the imperative programming languages).

Examples

- if statement
- for statement
- while statement
- switch statement
- ...

Assignment statements

The content of a variable can be changed via assignment statements as follows,

In [1]:

```
x = 6
y = -5
xy = 'Hello World'
```

Print statements

There are multiple ways to join values for printing out.

In [2]:

```
print (x+y, xy)
```

```
1 Hello World
```

Note that str() converts an intenger value into a string value.

In [3]:

```
print (str(x+y) + " " + xy)
```

```
1 Hello World
```

Note that %s is a template holder for string value, %d is for integer value, %f is for float number.

In [4]:

```
print("%d %s" % (x+y, xy))
```

```
1 Hello World
```

Do you know?

Multiple variables can be assigned with the same value.

In [5]:

```
x = y = 1
```

In [6]:

```
print (x,y)
```

```
1 1
```

Do you know Types?

At times, we need to check the types of the given variables. Types are static information of the possible run-time value, e.g. `int` , `string` , `boolean` and `float` etc.

Though Python is not strict about types, it is a good habit to ensure the types are matching, e.g. we don't want ot add a string value to float value, which may yield errors or undesired outcome.

To check for type of given value, we can use the built-in function `type()` . For example

```
type(first_name)
```

Operators

Arithmetic Operators

Symbol	Task Performed
+	Addition
-	Subtraction
/	division
%	mod
*	multiplication
//	floor division
**	to the power of

In [7]:

```
1+2
```

Out[7]:

```
3
```

In [8]:

```
2-1
```

Out[8]:

```
1
```

In [9]:

```
1*2
```

Out[9]:

```
2
```

Division (/) always returns a float for Python 3.7

In [10]:

```
1/2
```

Out[10]:

```
0.5
```

Floor division (//) will discard the fractional part

In [11]:

```
1//2
```

Out[11]:

```
0
```

In [12]:

```
2.8//2.0
```

Out[12]:

```
1.0
```

The % operator returns the remainder of the division

In [13]:

```
15%10
```

Out[13]:

```
5
```

Relational Operators

These operators define/test the relation between 2 entities, providing a boolean result stating whether the test performed is True or False.

Do you know Boolean Values?

There are only two Boolean values.

- `True`
- `False`

They are "printable".

```
>>> True
True
>>> print(False)
False
```

Symbol	Test Performed
==	True, if it is equal
!=	True, if not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Note the difference between an assignment statement using (=) and the relational operators (==)

In [14]:

```
z = 1
```

In [15]:

```
z == 1
```

Out[15]:

```
True
```

In [16]:

```
z > 1
```

Out[16]:

```
False
```

Bitwise Operators

Symbol	Task Performed
&	Logical And
|	Logical OR
^	XOR
~	Negate
>>	Right shift
<<	Left shift

In [17]:

```
000
001
010
011
100

a = 2 #10
b = 3 #11
```

In [18]:

```
print (a & b)
print (bin(a&b))
```

```
2
0b10
```

In [19]:

```
5 >> 1
```

Out[19]:

```
2
```

0000 0101 -> 5

Shifting the digits by 1 to the right and zero padding

0000 0010 -> 2

In [20]:

```
5 << 1
```

Out[20]:

```
10
```

0000 0101 -> 5

Shifting the digits by 1 to the left and zero padding

0000 1010 -> 10

Built-in Functions

Python comes loaded with pre-built functions

Conversion from one system to another

Conversion from hexadecimal to decimal is done by adding prefix **0x** to the hexadecimal value or vice versa by using built in **hex( )**, Octal to decimal by adding prefix **0** to the octal value or vice versa by using built in function **oct( )**.

In [21]:

```
hex(170)
```

Out[21]:

```
'0xaa'
```

In [22]:

```
0xAA
```

Out[22]:

```
170
```

In [23]:

```
oct(8)
```

Out[23]:

```
'0o10'
```

In [24]:

```
0o10
```

Out[24]:

```
8
```

int( ) accepts two values when used for conversion, one is the value in a different number system and the other is its base. Note that input number in the different number system should be of string type.

In [25]:

```
print (int('010',8))
print (int('0xaa',16))
print (int('1010',2))
```

```
8
170
10
```

int( ) can also be used to get only the integer value of a float number or can be used to convert a number which is of type string to integer format. Similarly, the function **str( )** can be used to convert the integer back to string format

In [26]:

```
print (int(7.7))
print (int('7'))
```

```
7
7
```

Also note that function **bin( )** is used for binary and **float( )** for decimal/float values. **chr( )** is used for converting ASCII to its alphabet equivalent, **ord( )** is used for the other way round.

In [27]:

```
chr(98)
```

Out[27]:

```
'b'
```

In [28]:

```
ord('b')
```

Out[28]:

```
98
```

Simplifying Arithmetic Operations

round( ) function rounds the input value to a specific number of decimal places or to the nearest integer.

In [29]:

```
print (round(5.6231))
print (round(4.55892, 2))
```

```
6
4.56
```

complex( ) is used to define a complex number and **abs( )** outputs the absolute value of the same.

In [30]:

```
c =complex('5+2j')
print (abs(c))
```

```
5.385164807134504
```

divmod(x,y) outputs the quotient and the remainder in a tuple (you will be learning about it in the further chapters) in the format (quotient, remainder).

In [31]:

```
divmod(9,2)
```

Out[31]:

```
(4, 1)
```

isinstance( ) returns True, if the first argument is an instance of that class. Multiple classes can also be checked at once.

In [32]:

```
print (isinstance(1, int))
print (isinstance(1.0,int))
print (isinstance(1.0,(int,float)))
```

```
True
False
True
```

pow(x,y,z) can be used to find the power x^y also the mod of the resulting value with the third specified number can be found i.e. : (x^y % z).

In [33]:

```
print (pow(3,3))
print (pow(3,3,5))
```

```
27
2
```

range( ) function outputs the integers of the specified range. It can also be used to generate a series by specifying the difference between the two numbers within a particular range. The elements are returned in a list (will be discussing in detail later.)

In [34]:

```
print (range(3))
print (range(2,9))
print (range(2,27,8))
```

```
range(0, 3)
range(2, 9)
range(2, 27, 8)
```

If-else

To define decision, we need to use if-else statement. An if-else statement has three components, namely,

1. conditional expression
2. then branch
3. else branch

```
if first_name == "Kenny":                    # conditional expression
    print("I know %s." % first_name)         # then branch
else:
    print("I don't know %s."% first_name)    # else branch
```

Note that the else branch is optional.

Exercise

What is the output of the following Python program?

```
i = 1
if (i / 2 >= 0.5):
    print(" %d / 2 is greater than or equal to 0.5" % (i))
else:
    print(" %d / 2 is less than 0.5" % (i))
```

In [35]:

```
# todo: Exercise
i = 1
if (i / 2 >= 0.5):
    print(" %d / 2 is greater than or equal to 0.5" % (i))
else:
    print(" %d / 2 is less than 0.5" % (i))
```

```
 1 / 2 is greater than or equal to 0.5
```

Conditional expression

Besides the if-else conditional statement, there is an if-else conditional expression in Python. Since it is an expression, it is meant to be used in place of the right side of the assignment, or some function application argument.

```
x = 1 / y if not (y == 0) else 0
```

which says, `x` will take the value of `1/y` as long as `y` is not zero, if `y` is zero, we assign `0` to `x`.

This conditional expression comes in handy in combination with the functional programming features mentioned later this practical.

None value

There is a very special value in Python, `None`, which denotes some undefined value or sometimes a less-disruptive way of error.

For example,

We define a safe division function,

```
def divide(x,y):
    if y == 0:
        return None
    else:
        return x / y
```

So now when we use `divide()` with some values, we should always check whether the returned value is `None`.

```
e = 10
d = 100
r = divide(e, d)

if r is None:
    print("division by zero error!")
else:
    print("the result is %d " % r)
```

Exercise

Can you re-define the `divide()` function using the conditional expression instead of the conditional statement?

In [36]:

```
# todo: Exercise
x=10
y=100
result = x/y if not (y == 0) else None
result
```

Out[36]:

```
0.1
```

For loop and while loop

In order to repeat some instructions for a fixed number or an indefinite number times, we need loop.

We use for-loop when we know the number of times that we want to repeat. (Though, we could also do that via while-loop, it is just more readable using for-loop.)

For-loop consists of an iteration declaration, and the loop body (the statements we want to repeat).

```
sum = 0
for i in range(0,10): # iteration declaration
    sum = sum + i     # body
```

Note `range(x,y)` is a built-in function which returns an ascending sequence of numbers starting from `x` ending with `y-1`.

A while-loop consists of a loop condition, and the loop body (the statements we want to repeat).

```
sum = 0
i = 0
while i < 10:     # loop condition
    sum = sum + i # body
    i = i + 1
```

While-loop is a more general and explicit form, which handles cases that we do not know explicitly the number of loops in advance. For instance, we want to keep prompting the user to key in the username and password, until the correct ones are entered.

```
uname = input("user name:")
pw    = input("password:")
while not(is_matched(uname,pw)):
    uname = input("user name:")
    pw    = input("password:")
```

Note that the above code can't run unless we provide the defintion of `is_matched`, which we omitted here.

Exercise

Write a Python program using for-loop to sum out all even numbers between 0 and 100. You can achieve the same result using while-loop. Please demonstrate and compare the two implementation.

Hint: to test whether a number `n` is even, you may use `n % 2 == 0`.

In [37]:

```
# todo: Exercise
sum1=0
sum2=0
for i in range(100):
    if i%2 == 0:
        sum1 = sum1 + i
k = 0;
while k < 100:
    if k%2 == 0:
        sum2 = sum2 + k
    k += 1

print (sum1)
print (sum2)
```

```
2450
2450
```

Functions

Functions are named code blocks which we would like to define once and re-use for multiple times. It is the first level of abstraction away implementation details to keep code simple, readable and resuable.

A function consists of a name, a set of formal arguments, and the function body. In the function body, the result of the computation might be returned.

```
def sum(x,y): # sum is the name, x and y are parameters
    return x + y # function body

def print_twice(mesg):
    print(mesg)
    print(mesg) # no return results
```

To call / to invoke a function, we need to use the function name in combination with the *actual* arguments.

```
print_twice(str(sum(10, 102)))
```

Exercise

Define and execute the `sum` and `print_twice` function.

In [38]:

```
# todo: Exercise
def sum(x,y): # sum is the name, x and y are parameters
    return x + y # function body

def print_twice(mesg):
    print(mesg)
    print(mesg) # no return results

print_twice(str(sum(10, 102)))
```

```
112
112
```

Recursive functions

A recursive function a function that calls itself in its body.

e.g.

```
    def factorial(n):
        if n <= 0:
            return 1
        else:
            return factorial(n-1)*n
```

Recursion functions are closer to their mathematical formulation.

Exercise

Can you define another version of `factorial` function that use a for-loop instead of recursion?

In [39]:

```
# todo: Exercise
def factorial1(n):
    if n <= 0:
        return 1
    else:
        return factorial1(n-1)*n

def factorial2(n):
    result = 1;
    for num in range(1,n+1):
        result = result * num
    return result

print(factorial1(4))
print(factorial2(4))
```

```
24
24
```

Higher order functions

A function is a higher order function if

- it takes another function as its formal argument, or
- it return another function as its result.

```
def times2(x):
    return x * 2

def apply_twice(f,x):
    return f(f(x))

apply_twice(times2, 10)
```

Exercise

Define the above functions and observe the output

In [40]:

```
# todo: Exercise
def times2(x):
    return x * 2

def apply_twice(f,x):
    return f(f(x))

apply_twice(times2, 10)
```

Out[40]:

```
40
```

List

A list is a data structure which denotes a collection of values. A list may have zero, one or more values. For instance,

```
ns = [1,2,3,4]
```

The size of the list can be computed using the builtin `len()` function.

```
len(ns)
```

To access an element in the list, we can use the index operation, `list_name[ index ]` , e.g.

```
ns[0] # access the first element in the list.
```

To append an element to a list, we can call the `.append()` method. e.g.

```
ns.append(5) # append 5 to the list as the 5th element.
```

To concatenate a list to another, we use the `+` operator, e.g.

```
ns + ns # which yields [1,2,3,4,5,1,2,3,4,5]
```

To reverse a list,

```
ns[::-1]
```

To extract a sub list from a list

```
ns[2:4] # extract the 3rd and the 4th elements, e.g. ns[2], and ns[3].
```

To iterate through all the elements in a list, we can use a for-loop.

```
for n in ns:
    print(n)
```

Exercise

Implement a sum_sq (sum of squares) function, which sums up all the squares of the elements in a list.

In [44]:

```
# todo: Exercise
ns = [1,2,3,4]

#for n in ns:
#    print(n)
#for n in range(len(ns)):
#    print(ns[n])

def sum_sq(x):
    total = 0
    for n in x:
        total = total + n*n
    return total

print(sum_sq(ns))

#"""
#import numpy as np
## creating an array using
## arrange method
#arr1 = np.array([1,2,3,4])

## iterating an array
#for a in np.nditer(arr1):
#    print(a)
#"""
```

```
30
```

If we copy a list with the equal sign only like this my_list_copy = my_list, you'll have the reference copied in the my_list_copy variable instead of the list values. So, if you want to copy the actual values, you can use the list(my_list) function or slicing [:].

Functional programming with map and reduce

Python supports a limited form of functional programming. Functional programming allows functions to be passed in and returned as value. Functions can be named or anonymous. This allows code to be concise and easier for verification and proving for correctness. For instance, we can redefine the `times2` function using lambda expression

```
times2_fun = lambda x: x*2
```

`map(f,l)` takes a function `f` and a list `l`, and apply `f` to every element in `l`.

```
list(map(times2_fun, ns))
```

yields a list of

```
[2,4,6,8]
```

Note: prior Python 3, `map()` returns a list. Since Python 3, `map()` returns an iterator of sequence to support laziness. To convert the object back to a list, a call of `list()` on the result is required.

Alternatively, we can use the lambda function directly

```
list(map(lambda x: x*2, ns))
```

`reduce(g,l)` takes a binary function `g` and a list `l`, and aggregates all elements in `l` with `g`.

For example,

```
from functools import reduce
reduce(lambda x,y: x+y, ns)
```

yields

```
10
```

Exercise

implement `sum_sq` using `map`, `reduce` and lambda expressions.

In [45]:

```
# todo: Exercise
ns = [1,2,3,4]

#square_fun = lambda x: x*x
#arr2 = list(map(square_fun,ns))

#from functools import reduce
#reduce(lambda x,y: x+y, arr2)

from functools import reduce
reduce(lambda x,y: x+y, list(map(lambda x: x*x,ns)))
```

Out[45]:

```
30
```

Functional programming (often abbreviated FP) is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects. Functional programming is declarative rather than imperative, and application state flows through pure functions. Contrast with object oriented programming, where application state is usually shared and colocated with methods in objects.

String

String is a special class of values in Python. On the high-level, it denotes a sequence of characters. It behaves like a character list.

```
s = "hello world"
length(s) # length of the string
s[0] # first character in the string
s[::-1] # reverse of the string
s + s # string concatenation
```

Besides all the functions inherits from list, we find some additional string functions which are very useful.

- Case conversion

  ```
  s.upper() # return a new string which is the upper case version of the original string
  ```

- Split by delimiters

  ```
  s.split(" ") # return a list of strings, ["hello", world"], " " is the delimintor.
  ```

- Removal of trailing space

  ```
  s.strip()
  ```

Dictionary

A Dictionary in Python is a look up table, consists of key-value pairs.

`{}` defines an empty dictionary.

```
d = {}
```

`d[key] = value` assigns a value to a slot in dictionary `d` indexed by `key`.

```
d["k1"] = 1
```

If the key already exists, the existing value will be overwritten.

`key in d` is a boolean expression which tests whether the key `key` is in dictionary `d`.

```
if "k1" in d:
    print ("k1 is in the dictionary")
```

`d[key]` returns the value store in `d` indexed by `key` if exists, otherwise, a Key Error will be thrown.

```
d["k1"]
```

Note that the values can be of any type. Only scalar values can be used as keys.

If we want to visit all the values in the dictionary, we can turn it into a key-value pair list and use for-loop to iterate over it.

```
for key, val in d.items():
    print(key, val)
```

Exercise

Implement a `wc` function which takes a string of text and counts and prints out the numbers of occurances of each word appearing in the text. A test case as follows,

```
wc('twinkle twinkle little star how i wonder what you are')
```

yields

```
little : 1
star : 1
twinkle : 2
i : 1
what : 1
how : 1
are : 1
you : 1
wonder : 1
```

In [46]:

```
# todo: Exercise

def wc(s):
    d = {}
    array1 = s.split(' ')
    for word in array1:
        if word in d:
            d[word] = d[word] + 1
        else:
            d[word] = 1

    for key, val in d.items():
        print(key,':', val)

wc('twinkle twinkle little star how i wonder what you are')
```

```
twinkle : 2
little : 1
star : 1
how : 1
i : 1
wonder : 1
what : 1
you : 1
are : 1
```

File I/O

To read the content from a file, (assuming it is a text file),

1. we use `open(filename,'r')` to initialize the file reader.
2. use for-loop to iterate through the content. (or while loop with read()).
3. `close()` the file reader when we are done.

To write the content to a file, (assuming it is a text file),

1. we use `open(filename,'w')` to initialize the file writer.
2. use `write()` to write content to file.
3. `close()` the file writer when we are done.

For instance, the following code copy the content from file a to file b.

```
def filecp(fileA, fileB):
    with open(fileA, 'r') as fa: # same as fa = open(fileA, 'r'), but it's better in excep
tion handelling
        with open(fileB, 'w') as fb:
            for l in fa:
                fb.write(l)
        fb.close()
    fa.close()
```

for more reference refer to [https://docs.python.org/3/tutorial/inputoutput.html (https://docs.python.org/3/tutorial/inputoutput.html)]

Exercise

Make use of the knowledge of the implementation of `wc()` and the Python File I/O api, implement a `wc_file()` function which is performing the same word count operation, but instead of taking a string as input, `wc_file()` should take a file name as input, count all the word occurances in the file and print out the result.

In [1]:

```
# todo: Exercise

def wc_file(fileA):
    with open(fileA, 'r') as fa:
        d = {}
        for l in fa:
            array3 = l.rstrip().split(' ')
            for word in array3:
                if word in d:
                    d[word] = d[word] + 1
                else:
                    d[word] = 1
        for key, val in d.items():
            print(key,':', val)
    fa.close()

wc_file("txtC.txt")
```

```
twinkle : 4
little : 2
star : 2
how : 2
i : 2
wonder : 2
what : 2
you : 2
are : 2
```

Tuple

Similar to list, tuple defines a collection of data values. Unlike list, the size of a tuple is fixed and not extendable.

```
t = (1,2)

print(t[0])
```

We can also "pattern match" a tuple value by putting the tuple pattern on the left hand side of the assignment statement. For instance

```
(x,y) = t
print(x)
```

Exercise

Define some tuple on your own and try to access its content.

In [48]:

```
# todo: exercise

t = (1,2)

print(t[0])

(x,y) = t
print(x)
```

```
1
1
```

Class and Objects

Object oriented programming allows us to define modula and resulable codes. The main idea is to pack data and method associated with the data into objects.

```
x = str("abc") # same as x = "abc"
x.upper() # .upper() is a method associated with string object x.
```

Objects are instantiated from a class, which serves like a blue print / template.

A class may inherit from another class. This allows features to be extended.

```
class Geometry(object):
    def __init__(self,_x,_y):
        self.x = _x
        self.y = _y

class Circle(Geometry):
    def __init__(self,_x,_y,_r):
        Geometry.__init__(self,_x,_y)
        self.r = _r

class Rectangle(Geometry):
    def __init__(self,_x,_y,_w,_h):
        Geometry.__init__(self,_x,_y)
        self.w = _w
        self.h = _h

circ = Circle(0,0,10)
circ.r
rect = Rectangle(3,-2,5,10)
rect.w
```

Exercise

Modify the above Geometry, Circle and Rectangle example. Add an overridden function call `.circumference()`. Note that the circumference calculation for circles is $$C = radius * 2 * \pi$$ π in Python is `math.pi`

The circumference calculation for rectangle is $$C = (width + height) * 2$$

In [1]:

```
# todo: Exercise
import math
class Geometry(object):
    def __init__(self,_x,_y):
        self.x = _x
        self.y = _y
    def circumference(self):
        return None

class Circle(Geometry):
    def __init__(self,_x,_y,_r):
        Geometry.__init__(self,_x,_y)
        self.r = _r
    def circumference(self):
        return self.r*2*math.pi

class Rectangle(Geometry):
    def __init__(self,_x,_y,_w,_h):
        Geometry.__init__(self,_x,_y)
        self.w = _w
        self.h = _h
    def circumference(self):
        return (self.w+self.h)*2

circ = Circle(0,0,10)
circ.r
rect = Rectangle(3,-2,5,10)
rect.w

print(circ.circumference())
print(rect.circumference())
```

```
62.83185307179586
30
```