1. Python Basic

1.1 Hello world

We will use the print() function to print our first python program: hello world.

Here, 'hello world' is a string, a sequence of characters. In Python, strings are enclosed in single quotes, double quotes, or triple quotes.

In [1]:
```
print('hello world')
print("hello world")
# triple quotes allows the string to span multiple lines
print("""hello

world""")
```

1.2 Comments

Comments are very important while writing a program. They explains the codes in lines for programmers to better understand a program.

In Python, we use the hash (#) symbol to start writing a comment.Python Interpreter ignores comments during excution.

- # for one line

We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line. Another way of doing this is to use triple quotes, either ''' or """.

- ''' ''' for multiple lines

In [2]:
```
print("hello world") #this is a comment
```

In [3]:
```
"""
triple quotes for
multiple
lines


"""
print("hello world")
```

```
hello world
```

1.3 Packages and modules

Python has packages and modules. A module is a file containing Python code. A package is like a directory that contains sub-packages and modules. A library contains multiple modules.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Python package can have sub-packages and modules.

To use these packages and modules in Python:

- import
- from ... import ...

In [4]:
```
import os # import os module
os.getcwd() # get the current working directory
```

Out[4]:
```
'D:\\lab\\code'
```

In [5]:
```
from os import getcwd # import getcwd method from os module
getcwd()
```

Out[5]:
```
'D:\\lab\\code'
```

If a package name is too complex to call, you can give it an alias.

In [6]:
```
import platform

platform.platform()
```

Out[6]:
```
'Windows-10-10.0.19041-SP0'
```

In [7]:
```
import platform as pf

pf.platform()
```

Out[7]:
```
'Windows-10-10.0.19041-SP0'
```

1.4 Variables

Variables are containers for storing data values. A variable is a named location used to store data in the memory.

Declaring and assigning value to a variable:

In [8]:
```
a = 1
print(a)
```

```
1
```

The value of a variable can be changed later in the program.

In [9]:
```
a = 2
print(a)
```

```
2
```

1.5 Naming rules and keywords

1.5.1 Naming rules

Variable names should have a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).

In [10]:
```
aA9_ = 1
```

Create a name that makes sense.

In [11]:
```
wasd = 100

price = 100
```

If you want to create a variable name having two words, use underscore to separate them.

In [12]:
```
number_of_cats = 6
```

Don't start a variable name with a digit.

In [13]:
```
1a = 1 # incorrect variable names
```

```
  File "<ipython-input-13-80e200d7b094>", line 1
    1a = 1 # incorrect variable names
     ^
SyntaxError: invalid syntax
```

1.5.2 Keywords

Keywords are the reserved words in Python.They cannot be used as ordinary identifiers and must be spelled exactly.

There are 35 keywords in Python 3.7.

In [58]:
```
def = 1 # when a variable name is a keyword
```

```
  File "<ipython-input-58-e1d06c92934c>", line 1
    def = 1 # when a variable name is a keyword
        ^
SyntaxError: invalid syntax
```

In [59]:
```
import keyword
keyword.kwlist
```

Out[59]:
```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

In [60]:
```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

1.6 Switching values

In [61]:
```
a = 1
b = 2
print(a)
print(b)
```

```
1
2
```

In [62]:
```
a,b = b,a
print(a)
print(b)
```

```
2
1
```

1.7 Excution Order in Python

From top to bottom:

In [63]:
```
x = 0
print(x+y)
y = 1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-63-e03f8af417bd> in <module>
      1 x = 0
----> 2 print(x+y)
      3 y = 1

NameError: name 'y' is not defined
```

1.8 Indentation

A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line.

The number of space to be indented is up to you, but it must be consistent throughout that block.

Indentation makes Python code look neat and clean. This also makes Python code look similar and consistent.

In [64]:
```
def func1():
   a = 1
   b = 1

def func2():
    a = 2
    b = 2
```

1.9 Global Variables and Local Variables

Global variables are variables declared in global scope (code block without indentation).

Local variables are variables declared in local scope (code block with indentation).

In [65]:
```
a = 1 # Global Variables

def func(): # define a function
    f = 2 # Local Variables


c = 3

print(a+c)
```

```
4
```

In [66]:
```
print(a+f)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-66-e5cacae4cdcc> in <module>
----> 1 print(a+f)

TypeError: unsupported operand type(s) for +: 'int' and 'function'
```

1.10 Standard I/O

To read from standard input, use input() function.

To write to standard ouput, use print() function.

In [67]:
```
s = input("input: ")
print(s)
```

```
input: I love Python
I love Python
```

1.11 Python Basic Functions

1.11.1 help() function

The Python help function is used to display the documentation of modules, functions, classes, keywords etc.

In [68]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

1.11.2 id() function

id function returns the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.

In [69]:
```
a = 1
id(print)
```

Out[69]:
```
2693858468152
```

1.11.3 type() function

type function return the type of an object.

In [70]:
```
a = 1
type(a)
```

Out[70]:
```
int
```

1.11.4 del() function

del function deletes an object

In [71]:
```
a = 1
del(a)
print(a)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-71-691d79be5171> in <module>
      1 a = 1
      2 del(a)
----> 3 print(a)

NameError: name 'a' is not defined
```

1.11.5 len() function

len function returns the length (the number of items) of an object.

In [72]:
```
len("hello")
```

Out[72]:
```
5
```

2. Python Data Types

2.1 Numeric: int, float, complex

Number Data Type in Python

Python supports integers, floating-point numbers and complex numbers.

Integers and floating points are separated by the presence or absence of a decimal point.

Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

Integers can be of any length, a floating-point number is accurate only up to 15 decimal places (the 16th place is inaccurate).

In [73]:
```
a = 5

print(type(a))
```

```
<class 'int'>
```

In [74]:
```
a = 5.0

print(type(a))
```

```
<class 'float'>
```

In [75]:
```
a = 5 + 6j
print(a + 6)

print(type(a))
```

```
(11+6j)
<class 'complex'>
```

2.2 Text: str

2.2.1 String

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn about Unicode from Python Unicode.

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

In [76]:
```
# create a string

s1 = "Χαίρετε Python"

s2 = 'Χαίρετε Python' # in a single quote or double-quotes

s3 = """Χαίρετε

       Python
    """ # triple quotes for multiline strings and docstrings
print(s1)
print(s2)
print(s3)
print(type(s1))
print(type(s2))
print(type(s3))
```

```
Χαίρετε Python
Χαίρετε Python
Χαίρετε

       Python

<class 'str'>
<class 'str'>
<class 'str'>
```

2.2.2 Escape Sequence

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An escape sequence starts with a backslash and is interpreted differently. If we use a single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes.

In [77]:
```
# to show the Double quote
s = "python\""
print(s)
```

```
python"
```

In [78]:
```
# to show the Backslash
s = "python\\"
print(s)
```

```
python\
```

In [79]:
```
# to start a new line
s = "py\nthon"
print(s)
```

```
py
thon
```

2.2.3 Raw String

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

In [80]:
```
s = r"C:\abc\abc\abc.txt"
print(s)
```

```
C:\abc\abc\abc.txt
```

2.2.4 Access Characters in a String

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use floats or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

In [81]:
```
s = "python"
s[0]
```

Out[81]: 'p'

In [82]:
```
s[1]
```

Out[82]: 'y'

In [83]:
```
s[-1]
```

Out[83]: 'n'

In [84]:
```
s[99]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-84-d35b3764b244> in <module>
----> 1 s[99]

IndexError: string index out of range
```

In [85]:
```
s[1.0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-e1d895c8c1d6> in <module>
----> 1 s[1.0]

TypeError: string indices must be integers
```

In [86]:
```
# python
s[0:3]
```

Out[86]:
```
'pyt'
```

In [87]:
```
# python
s[0:4:2]
```

Out[87]:
```
'pt'
```

In [88]:
```
# unchangeable
s[0] = "l"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-88-38542d03ab79> in <module>
      1 # unchangeable
----> 2 s[0] = "l"

TypeError: 'str' object does not support item assignment
```

2.2.5 String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python.

Joining of two or more strings into a single one is called concatenation. The + operator does this in Python.

The * operator can be used to repeat the string for a given number of times.

In [89]:
```
# + operator
a = "hello"
b = "world"
print(a+b)
```

```
helloworld
```

In [90]:
```
# * operator
a = "hello"
b = 2
print(a*b)
```

```
hellohello
```

2.2.6 Common Python String Methods

There are numerous methods available with the string object.

Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc.

In [91]:
```
# split
# str.split(str1)：split a string by str1
s = "python is cool"
s.split(' ')
```

Out[91]:
```
['python', 'is', 'cool']
```

In [92]:
```
# replace
# str.replace(str1,str2)：replace the str1 by str2
s = 'python'
s.replace('py','ABC')
```

Out[92]:
```
'ABCthon'
```

In [93]:
```
# lower and upper
# str.lower()：lower a string
# str.upper()：upper a string
s = "PYthon"
print(s.lower())
print(s.upper())
```

```
python
PYTHON
```

In [94]:
```
# join
# str.join(iter)：join the string by iter
"--".join(['python', 'is', 'cool'])
```

Out[94]:
```
'python--is--cool'
```

2.2.7 String formatting

In [95]:
```
# switch %s and %d with AI and 63
'My name is %s , age is %d' %('AI', 63)
```

Out[95]:
```
'My name is AI , age is 63'
```

2.3 Lists

Lists are mutable sequences, typically used to store collections of homogeneous items.

List = [obj1, obj2, ….]

In [96]:
```
l = [1,2,3] # creat a list
print(l)
```

```
[1, 2, 3]
```

In [97]:
```
# access items in a list
print(l[1])
print(l[0:2])
print(l[0:3:2])
```

```
2
[1, 2]
[1, 3]
```

In [98]:
```
# change items in a list
l[1] = 0
print(l)
```

```
[1, 0, 3]
```

2.3.1 Lists Operators

Same as string operators.

In [99]:
```
a = [1,2,3]
b = [4,5]
```

In [100]:
```
# * operator
print(a*2)
# + operator
print(a+b)
```

```
[1, 2, 3, 1, 2, 3]
[1, 2, 3, 4, 5]
```

2.3.2 Lists Methods

In [101]:
```
animals = ['cat', 'dog', 'bird']
# list.append(obj): add one item to the end of a list
animals.append('fish')#  add fish
print(animals)
```

```
['cat', 'dog', 'bird', 'fish']
```

In [102]:
```
# list.remove(obj): delete one or more items from a list
animals.remove('fish')# delete fish
print(animals)
```

```
['cat', 'dog', 'bird']
```

In [103]:
```
# list.insert(index, obj): insert an item at the defined index
animals.insert(1,'fish') # insert fish at index 1
print(animals)
```

```
['cat', 'fish', 'dog', 'bird']
```

In [104]:
```
# list.pop([index=-1]): removes and returns the last item if the index is not pro
print(animals.pop()) # removes the item at index -1 and returns the left items
print(animals)
```

```
bird
['cat', 'fish', 'dog']
```

In [105]:
```
# enumerate(sequence): adds counter to an iterable and returns it.
for i, x in enumerate(animals):
    print(i, x)
```

```
0 cat
1 fish
2 dog
```

In [106]:
```
# List sort
list1 = [12,45,32,55]
# list.sort(cmp=None, key=None, reverse=False): sorts the elements of a given lis
list1.sort() # sort list1
print(list1)
```

```
[12, 32, 45, 55]
```

In [107]:
```
# list.reverse(): reverses the elements of the list.
list1.reverse() # reverses list1
print(list1)
```

```
[55, 45, 32, 12]
```

In [108]:
```
# List Comprehension
squares = [x*2 for x in animals] # define and create lists based on existing list
print(squares)
```

```
['catcat', 'fishfish', 'dogdog']
```

2.3.3 Lists slicing and indexing

List indexing returns the index of the specified element in the list.

In [109]:
```
l = [[1,3],[4,5]]
l[1][0]
```

Out[109]:
```
4
```

List slicing returns a slice object that can be used to slice strings, lists, tuple etc.

list[start:end:step]:

- start: where slicing starts; Default to None if not provided.
- end: where slicing stops;
- step: determines the increment between each index for slicing. Defaults to None if not provided.

In [110]:
```
l = [0,1,2,3,4,5]
l[::2]
```

Out[110]:
```
[0, 2, 4]
```

In [111]:
```
l[1:4:2]
```

Out[111]:
```
[1, 3]
```

2.4 Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the enumerate() built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a set or dict instance).

(obj1,obj2.....)

In [112]:
```
# creat a tuple
t = (1,[1,2],"python")
print(t)
print(type(t))
```

```
(1, [1, 2], 'python')
<class 'tuple'>
```

In [113]:
```
# creat a tuple with an item
t1 = (5)
t2 = (5,) # “, ” is needed
print("t1",type(t1))
print("t2",type(t2))
```

```
t1 <class 'int'>
t2 <class 'tuple'>
```

In [114]:
```
# tuple indexing
t = (1,2,3)
print(t[1])
```

```
2
```

In [115]:
```
# tuple unchangeable
t[1] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-115-b8896bff046e> in <module>
      1 # tuple unchangeable
----> 2 t[1] = 0

TypeError: 'tuple' object does not support item assignment
```

2.5 Dictionaries

A dictionary is an unordered collection of items. Each item of a dictionary has a key:value pair.The keys within one dictionary are unique. If you store data using a key that is already in use, the old value associated with that key is forgotten. Extracting a value using a non-existent key will cause error.

Dictionaries are unordered, changeable and does not allow duplicates. A pair of braces creates an empty dictionary: {}.

{key:value}

In [116]:
```
# creat a dictionary in three different ways
x = {'food':'Spam','quantity':4,'color':'pink'}
x2 =dict(food='Spam',quantity=4, color='pink')
x3 = dict([("food", "Spam"),("quantity", 4),("color","pink")])

print(x)
print(x2)
print(x3)
```

```
{'food': 'Spam', 'quantity': 4, 'color': 'pink'}
{'food': 'Spam', 'quantity': 4, 'color': 'pink'}
{'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

2.5.1 Access items in dictionaries

In [117]:
```
# extract values in a dictionary
print(x["food"])
```

```
Spam
```

In [118]:
```
print(x["a"]) # report an error when extract a value using a non-existent key.
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-118-759f8566e7f5> in <module>
----> 1 print(x["a"]) # report an error when extract a value using a non-existe
nt key.

KeyError: 'a'
```

In [119]:
```
# extract values by get(key,["str"])
print(x.get("food"))
print(x.get("a")) # by get() method, returns none when extract a value using a no
print(x.get("a","cannot find a"))
```

```
Spam
None
cannot find a
```

In [120]:
```
# extract all keys
print(x.keys())
# extract all values
print(x.values())
# extract all key:value pairs
print(x.items())
```

```
dict_keys(['food', 'quantity', 'color'])
dict_values(['Spam', 4, 'pink'])
dict_items([('food', 'Spam'), ('quantity', 4), ('color', 'pink')])
```

In [121]:
```
# insert an item to a dictionary
x["x_key"] = "x_val"
print(x)
```

```
{'food': 'Spam', 'quantity': 4, 'color': 'pink', 'x_key': 'x_val'}
```

In [122]:
```
# change an item in a dictionary
x["x_key"] = 0
print(x)
```

```
{'food': 'Spam', 'quantity': 4, 'color': 'pink', 'x_key': 0}
```

2.6 Sets

A set is an unordered collection with no duplicate elements.

Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary

{obj1, obj2,....}

In [123]:
```
sample_set = {'Prince', 'Techs'}
sample_set2 = set(['Prince', 'Techs'])
print(sample_set)
print(sample_set2)
```

```
{'Techs', 'Prince'}
{'Techs', 'Prince'}
```

In [124]:
```
print('Data' in sample_set)    # The in keyword is used to check if a value is pr
print('Techs' in sample_set)
```

```
False
True
```

In [125]:
```
# set.add(obj): adds a given element to a set. If the element is already present,
sample_set.add('Data')
print(sample_set)
print(len(sample_set))
```

```
{'Techs', 'Data', 'Prince'}
3
```

In [126]:
```
# set.remove(obj)：removes an element from a set
sample_set.remove('Data')    # removes Data
print(sample_set)
```

```
{'Techs', 'Prince'}
```

In [127]:
```
list2 = [1,3,1,5,3]
print(list(set(list2)))# set has no duplicate elements, thus can be used to remov
```

```
[1, 3, 5]
```

In [128]:
```
sample_set = frozenset(sample_set) # unchangeable set
sample_set.add('cat')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-128-35f062032d89> in <module>
      1 sample_set = frozenset(sample_set) # unchangeable set
----> 2 sample_set.add('cat')

AttributeError: 'frozenset' object has no attribute 'add'
```

2.7 Operations on data

2.7.1 Deep copy and Shallow copy

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other.

copy.copy(x)

- Return a shallow copy of x.

copy.deepcopy(x[, memo])

- Return a deep copy of x.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

In [129]:
```
# data assignment
a = [1,2,3,[4,5]]
b = a # assign a to b
b[0]=99
print(a)
```

```
[99, 2, 3, [4, 5]]
```

In [130]:
```
# data assignment
a = [1,2,3,[4,5]]
b = a.copy() # shallow copy
b[0]=99
print(a)
```

```
[1, 2, 3, [4, 5]]
```

In [131]:
```
b[3][0]=99
print(a)
```

```
[1, 2, 3, [99, 5]]
```

In [132]:
```
import copy
# data assignment
a = [1,2,3,[4,5]]
b = copy.deepcopy(a) # deep copy
b[3][0]=99
print(a)
```

```
[1, 2, 3, [4, 5]]
```

2.7.2 Operators

In [133]:
```
# assignment operators
a=5
a+=1
print("a+=1:",a)
a-=1
print("a-=1:",a)
a*=2
print("a*=2:",a)
a/=2
print("a/=2:",a)
```

```
a+=1: 6
a-=1: 5
a*=2: 10
a/=2: 5.0
```

In [134]:
```
# comparison operators
print(1>=2)
print(1<=2)
print(1!=2)
print(1==2)
```

```
False
True
True
False
```

In [135]:
```
# logical operators
True and False
```

Out[135]: False

In [136]:
```
True or False
```

Out[136]: True

In [137]:
```
not True
```

Out[137]: False

In [138]:
```
# membership operators
a = [1,2,3]
1 in a
```

Out[138]: True

In [139]:
```
4 not in a
```

Out[139]: True

In [140]:
```
# Identity operators
a = [1,2]
b = a
c = a.copy()

print(a is b)
print(a is c)
print(a == b == c)
```

```
True
False
True
```

3. Control Flow Statement

The if, while and for statements implement traditional control flow constructs.

3.1 if statement

The if statement is used for conditional execution.

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true; then that suite is executed (and no other part of the if statement is executed or evaluated). If all expressions are false, the suite of the else clause, if present, is executed.

There can be zero or more elif parts, and the else part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation.

An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

determine the grade by scores

In [141]:
```
score = input("Please enter a score: ") # here input is a string
```

```
Please enter a score: 78
```

In [145]:
```
type(score)
```

Out[145]:
```
float
```

In [146]:
```
score = float(score) # transform the string to a floating-point number
```

In [147]:
```
if 100>=score>=60:
    print("A")
else:
    print("F")
```

```
A
```

In [148]:
```
if 100>=score>=90:  # set conditions by if-elif-else statements
    print("A")
elif 90 > score >= 80:
    print("B")
elif 80>score>=60:
    print("C")
elif 60>score>=0:
    print("F")
else:
    print("Please enter a correct score ")
```

```
C
```

The conditions can be True or False. 0 and None value are also False, while the other values are Ture.

In [149]:
```
if 0:
    print("F")
if 1:
    print("T")
if "python":
    print("python")
if None:
    print("None")
```

```
T
python
```

3.2 Loops

3.2.1 while loop

The while statement is used for repeated execution as long as an expression is true. It's used when we don't know the number of times to iterate beforehand.

A break statement executed in the first suite terminates the loop without executing the else clause’s suite.

A continue statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

In [150]:
```
i = 0
while i<9:
    i+=1
    print(i)
else:
    print("loop ends")
```

```
1
2
3
4
5
6
7
8
9
loop ends
```

In [151]:
```
i = 0
while i<9:
    i+=1
    if i == 3:
        print("Skip this loop")
        continue # skips the rest of the suite and goes back to testing the expre
    print(i)
else:
    print("loop ends")
```

```
1
2
Skip this loop
4
5
6
7
8
9
loop ends
```

In [152]:
```
i = 0
while i<9:
    i+=1
    if i == 5:
        print("Terminates the loop")
        break  # terminates the loop without executing the else clause’s suite
    print(i)
else:
    print("loop ends")
```

```
1
2
3
4
Terminates the loop
```

3.2.2 for loop

The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object.

In [153]:
```
times = ["first","second","third"]
for i in times:
    print(i)
```

```
first
second
third
```

Else statement can be added in the end of for statement. It will be excused when the loop normally ends.

In [154]:
```
for i in times:
    print(i)
else:
    print("The loop ends")
```

```
first
second
third
The loop ends
```

3.2.3 Nested Loops

Loops can be nested in Python. A nested loop is a loop that occurs within another loop.Nested loops go over two or more loops.

In [155]:
```
# write a multiplication table
for i in range(1,10): # a outer loop
    for j in range(1,i+1): # a inner loop
        # string formatting
        print("%dX%d=%-2d"%(j,i,j*i), end=" ")
    print()
```

```
1X1=1
1X2=2  2X2=4
1X3=3  2X3=6  3X3=9
1X4=4  2X4=8  3X4=12 4X4=16
1X5=5  2X5=10 3X5=15 4X5=20 5X5=25
1X6=6  2X6=12 3X6=18 4X6=24 5X6=30 6X6=36
1X7=7  2X7=14 3X7=21 4X7=28 5X7=35 6X7=42 7X7=49
1X8=8  2X8=16 3X8=24 4X8=32 5X8=40 6X8=48 7X8=56 8X8=64
1X9=9  2X9=18 3X9=27 4X9=36 5X9=45 6X9=54 7X9=63 8X9=72 9X9=81
```

4. Functions in Python

4.1 Functions

In addition to built-in functions, we can define our own functions by def statemesnt and lambda statement.

In [156]:
```
def func():
    print("hello")
    print("world")

func()
```

```
hello
world
```

There are multiple ways to pass data to a function:

- positional arguments
- default arguments
- keyword arguments
- arbitrary positional arguments
- arbitrary keyword arguments

In [157]:
```
def f(a=128, b=2, *args, **kwargs):
    print("a: %d" %(a))
    print("b: %d" %(b))
    print("args:",args)
    for key, value in kwargs.items():
        print ("%s is %s" %(key, value))
    print()

# default arguments
f()
```

```
a: 128
b: 2
args: ()
```

In [158]:
```
# positional arguments
f(0,4)
```

```
a: 0
b: 4
args: ()
```

In [159]:
```
# keyword arguments
f(b=4,a=0)
```

```
a: 0
b: 4
args: ()
```

In [160]:
```
'''
def f(a=128, b=2, *args, **kwargs):
    print("a: %d" %(a))
    print("b: %d" %(b))
    print("args:",args)
    for key, value in kwargs.items():
        print ("%s is %s" %(key, value))
    print()
'''

# arbitrary positional arguments
f(1,2,3,4,5)
```

```
a: 1
b: 2
args: (3, 4, 5)
```

In [161]:
```
'''
def f(a=128, b=2, *args, **kwargs):
    print("a: %d" %(a))
    print("b: %d" %(b))
    print("args:",args)
    for key, value in kwargs.items():
        print ("%s is %s" %(key, value))
    print()
'''

# arbitrary keyword arguments
f(cat='animal', dog='animal', cheese='food')
```

```
a: 128
b: 2
args: ()
cat is animal
dog is animal
cheese is food
```

return value

In [162]:
```
# single return value
def func(a,b):
    return a+b

a = func(2,3)
print("a:",a)
```

```
a: 5
```

In [163]:
```
# multiple return value
def func():
    return 1,2,3

a = func()
print("a:",a)
```

```
a: (1, 2, 3)
```

Define a function that takes a number as argument, and returns the fibonacci sequence. The length of the returned sequence is decided by the argument.

In [164]:
```
def fibs(num):
    result = [0,1]
    for i in range(2,num):
        a = result[i-1] + result[i-2]
        result.append(a)
    return result

fibs(5)
```

Out[164]:
```
[0, 1, 1, 2, 3]
```

4.2 Anonymous function

lambda expressions are used to create anonymous functions. lambda expression yields an unamed function object that behaves like:

- def (parameters):
  - return expression

In [165]:
```
a = lambda x,y:x+y

a(1,2)
```

Out[165]:
```
3
```

4.3 Higher-order function

A higher-order function is a function that takes functions as arguments, or return a function.

In [166]:
```
# zip function
a = [1,2,3]
b = ['a','b','c']
for x in zip(a,b):
    print(x)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

In [167]:
```
# map
print(*map(lambda x:x*x, [1,2,3]))
```

```
1 4 9
```

In [168]:
```
# filter
print(*filter(lambda x: x%2==1, [1,2,3]))
```

```
1 3
```

In [169]:
```
# sorted
sorted([('b',2),('a',1),('c',3),('d',4)], key=lambda x:x[1])
```

Out[169]:
```
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

5. Object Oriented Programming

Object-oriented programming is a programming paradigm based on the concept of "objects", which can contain data (variables) and methods (functions).

Advantages of Object-oriented programming:

- Code reusability
- Easier to maintain
- Better productivity
- Easier to extend

5.1 class

Python classes provide all the standard features of Object Oriented Programming:

- Encapsulation
- Inheritance
- Abstraction

- Polymorphism

In [170]:
```
# create a class that simulates a dog
class Dog():
    def __init__ (self, name, age):
        self.name = name
        self.age = age

    def sit(self):
        print(self.name.title()+"is now sitting")

    def run(self):
        print(self.name.title()+"is now running")
```

In [171]:
```
# instantiation
my_dog = Dog("Husky", 3)
my_dog.sit()
my_dog.run()
```

```
Huskyis now sitting
Huskyis now running
```

We can also add new data to an object

In [172]:
```
my_dog.food = 'cat food'
print(my_dog.food)
```

```
cat food
```

we can also re-define a class

In [173]:
```
class Dog(object):
    def __init__(self, name):
        self.name = name
        print("%s has been created"%(self.name))
my_dog = Dog("Goodog")
```

```
Goodog has been created
```

implement the str method

In [174]:
```
class Dog(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return "dog name: "+self.name
my_dog = Dog("Goodog")
print(my_dog)
```

```
dog name: Goodog
```

5.2 Encapsulation

Encapsulation can be used to hide the values of data inside an object, preventing unauthorized parties from direct accessing them.

In [175]:
```
class SimpleCounter:
    __secretCount = 0  # private variable
    publicCount = 0    # public variable
    def count(self):
        self.__secretCount += 100
        print(self.__secretCount)

counter = SimpleCounter()
counter.count()
print(counter.publicCount)
```

```
100
0
```

In [176]:
```
print(counter.__secretCount)  # cannot access private variable directly
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-176-1335881a4be1> in <module>
----> 1 print(counter.__secretCount)  # cannot access private variable directly

AttributeError: 'SimpleCounter' object has no attribute '__secretCount'
```

5.3 Inheritance

Inheritance allows us to define a class that inherits all the methods and data from another class.

In [177]:
```
class Parent: # parent class
    parentAttr = 100
    def __init__(self):
        print("instantiating parent")
    def parentMethod(self):
        print('parent method')
    def setAttr(self, attr):
        Parent.parentAttr = attr
    def getAttr(self):
        print("parent attribute: ", Parent.parentAttr)

class Child(Parent): # child class
    def __init__(self):
        print("instantiating child")
    def childMethod(self):
        print('child method')
    def getAttr(self):
        super().getAttr()
        print("child says hello!")


c = Child()          # instantiate child
c.childMethod()      # call child method
c.parentMethod()     # call parent method
c.setAttr(200)       # set attribute (parent method)
c.getAttr()          # call attribute (parent method)
```

```
instantiating child
child method
parent method
parent attribute:  200
child says hello!
```

In [178]:
```
c.childMethod()      # call child method
```

```
child method
```

5.4 Abstraction

An abstract class is a blueprint for other classes. We can create a set of methods that is required to be created within any child classes built from the abstract class.

The main goal of abstraction is to handle complexity by hiding details from the user.

The ABC module provides the infrastructure for defining abstract base classes (ABCs) in Python.

In [179]:
```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def eats_grass(self):
        pass

    def is_animal(self):
        return True

class Cat(Animal):
    def __init__(self):
        print("miew~~")
    # overriding abstract method
    def eats_grass(self):
        return False


miew = Cat()
print(miew.is_animal())
print(miew.eats_grass())
```

```
miew~~
True
False
```

5.5 Polymorphism

Polymorphism is the ability of an object to have many forms.

Polymorphism with functions

In [180]:
```python
print(len("hello python!"))

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

```
13
5
```

Polymorphism with classes, and inheritance

In [181]:
```
class Animal:
    def sound(self):
        pass

class Cat(Animal):
    def sound(self):
        print("miew")

class Bird(Animal):
    def sound(self):
        print("jiujiujiu")

class Fish(Animal):
    def sound(self):
        print("...")
```

In [182]:
```
kitty = Cat()
petty = Bird()
jelly = Fish()
my_pets = [kitty, petty, jelly]

for x in my_pets:
    x.sound()
```

```
miew
jiujiujiu
...
```

In [183]:
```
print(isinstance(kitty, Cat))
print(isinstance(kitty, Animal))
```

```
True
True
```

In [ ]:
```

```