Python Basics

Part 1

Identifiers, objects and assignment

- x = 10
- case sensitive
- any combination of letters, numerals and underscore characters (unicode characters)
- cannot begin with a numeral and cannot be reserved words
- reserved words - False, True, continue, if, else, while, for, None, etc.
- Dynamic typing: x can be used to store anything
- Aliases of an object: y = x then check the value of y after changing x and vice-versa
- Breaking the alias: x = x + 3 then check the value of y?

Built-in Classes

- support for literal form: x = 3.14 (creates an instance of float class) and 3.14 is a numeric (floating point) literal
- immutable classes - bool, int, float, tuple, str
- mutable classes - list, set, dict
- instantiation of the built in classes
  - bool( ) - allows creation from both boolean and non-boolean parameters - numbers other than 0 are interpreted as True and 0 is False, sequences (list, set, str) which are empty are evaluated as False and non-empty as True
  - int( ) - for arbitrary magnitude - 0b1011, 0o48, 0x6e are binary, octal and hexadecimal numbers; blank parameter returns 0, float parameter is truncated, appropriate string parameter (by default to base 10) converted to integer otherwise ValueError, for different base pass the base as second parameter e.g. int('7e', 16) which is 126
  - float( ) - fixed precision representation, no parameter returns 0.0, appropriate parameter returns a floating point value, even an appropriate string is ok

Sequences

- sequence types - list, str, tuple - 0 indexed, ordered elements; single character string is like a char type

- list is a sequence of references (so elements can be a combination of types unlike arrays in other languages); zero indexed; uses [ ] delimiters; [ ] is an empty list; list can be sequence of literals or any objects; list( ) is an empty list; list( ) constructor can accept any Iterable type (i.e. container types such as string, list, tuple, set, dict), list('cat') produces ['c','a','t']; l1 = list(l2) can produce a new list l2 from existing list l1

- tuple - like an immutable list; uses ( ) delimiters; tuple( ) creates an empty tuple; tuple(3,) produces a single element tuple

- str - immutable sequence of characters; based on unicode char set; single or double quotes, can be 'can\'t say' where \ is an escape sequence, similarly try 'can\'t say \\ t' \n \t are other escape sequences; unicode sequences can be included e.g. try 'around \u20B9 10' ; ''' or """ for multiline strings (useful for embedding comments)

Expressions, Operators, Precedence

- logical operators: not, and, or - these are short circuit operators - e.g.

l1 = [2,3,4]
b = 2==2 or l1[4]== 2
b = 2==3 and l1[4] == 2

- equality operators: is, is not, == , != the "is" checks for identity of objects i.e. if they are the same objects e.g. a is b - a may not be b but they may be equal in value i.e. a == b

a = (2,3)
b = a
a is b
c = (2,3)
a is c
a == c

- comparison operators: <, <=, >, >= for numeric it is obvious but technically it is done lexicographically, case sensitively for strings, for incompatible types exception is raised. Examples, compare two tuples (2,3) and (2,4) and (2,4,1), compare 2 and 'ab' (this will give TypeError)

- arithmetic operators: +, -, *, /, //, % - int with int is int, float with int is float; / is true division, // is mathematical floor of quotient, % is remainder of integer division; n / m gives q = n // m and r = n % m and always q * m + r = n irrespective of sign - try 29 / 4, 29 / -4, -29 / 4, -29 / -4 :: When the divisor m is positive, Python further guarantees that 0 ≤ r < m. When the divisor is negative, Python guarantees that m<r≤0

Operators - 2

- Sequence operators (for all sequence types - str, tuple, list): myseq[i], myseq[i:j], myseq[i:j:k], myseq1 + myseq2, myseq * n, x in myseq, x not in myseq; zero indexing, negative indexing, slicing notation (half-open intervals). Experiment with various options; lists can be modified using slicing notation, use of del, ==, !=, <,<=, >,>= also applicable where element by element comparison done until mismatch (i.e. based on lexicographical order)

```
l1 = [2,3,4,5,6,7]
l1[1:4] = [1]*3
del l1[4]
```

- Set operators: in, not in, ==, !=, <=, <, >, >= are subset set and proper subset operators, | is union operator, & is intersection operator, - is difference operator, ^ is xor operator (i.e. select elements which are in only one of the two sets). Experiment with == operator on sets with same elements in different order.

- Dict operators: d[k], d[k] =v, del d[k], k in d, k not in d, d1 == d2, d1 != d2

- Extended assignment operator: x += 1 (and for most binary operators such as *, /, -), for immutable types (number, str) reassignment takes place creating a new object; but for mutable type such as list existing object is modified: l2 = l1, l2 += [4,5] (now check for l1 after the extension done through alias l2) which is different from l2 = l2 + [4,5] here an extension and then reassignment of l2 takes place i.e. new object created (now check for l1).

Slicing operator more examples

```
l900 = [ ]
for i in range(97,123):
    l900.append(chr(i))
import random
random.shuffle(l900)
l900

l900[-10:-2:2]
l900[-10:2:-3]
```

Compound Expressions

- chained assignment: x = y = 0 (all assigned to right most value)
- chaining of comparison operators: 1 <= x + y <= 10 is same as (1 <= x + y) and (x+ y < = 10) this is done without computing intermediate value x + y twice.
- Operator precedence

Operator Precedence

- operator precedence from https://docs.python.org/3/reference/expressions.html#comparisons
- below gives lowest to highest precedence from left to right
- assignment, conditional, or, and, not, comparison operators (is, is not, in, not in, >,<,=,!,==),bitwise operators(&,|,^,<<,>>),arithmetic operators (DMAS), unary operator (+,-), exponentiation, function call, member access

```
l1 = [4,2,6]
l2 = [1]
[1,4,2,6] == l2 + l1

2+3**3

True and [1,4,2,6] == l2 + l1
False and [1] == l2[2]

x = 1
x+=3**2
x
x in l1+l2

a = 6+3 < 10
a
l1 = [2,3,4,5]
b = 2> l1.index(4)
b
l1 = [2,4,5]
a = 3 == l1.count(4)
a
```

Control Flow - I

- conditionals: if-elif-else
- non boolean types may be evaluated as booleans so can be used in conditionals
- nested ifs

```
a = ''
if a:
    print('not empty')
else:
    print('empty')
```

Control Flow - II

- Loops: while, for

```
l3 = [ ]
#l3 = [4,2,3,6,1]
j = 0
while j < len(l3) and l3[j] < 5:
    print(l3[j])
    j +=1
```

- what if we have this below:

```
l3 = [4,2,3,2,1]
j = 0
while  l3[j] < 5 and j < len(l3):
    print(l3[j])
    j +=1
```

Control Flow - III

- for loop

for element in iterable:

    do this body

- the element is freshly assigned each element of the iterable one by one
- the loop repeats until all elments are exhausted
- no need to manage a loop iteration index mechanism (as was required in a while loop)
- if the element is re-assigned it does not affect the loop operation

```
l1 = [2,3,4,5]
for e in l1:
    if e % 2 == 0:
        print(e)
    else:
        e += 1
        print(e)
```

- index based loops: for i in range(len(l1)): do this
- break and continue : applies to both while and for
- pass statement

Functions

- signature: def, name, parameters
- return statement: optional, multiple return statements, returning multiple values
- functions as objects and can be passed as parameters
- formal parameters and actual parameters
- scope and lifetime of variables inside function body
- more about functions and its advanced features to be covered later (variable number of positional and keyword arguments)
- difference between functions and methods (of objects)

```
def f1(x,y):
    z = x + y
    if x % 2 == 0:
        return z
    else:
        return z-1

def f2(x):
    if x% 2 == 0:
        return True
    return False

def f3(x):
    return x+2,x-2

def f4(f,x):
    return f(x)

print(f'f1: {f1(2,3)}')
print(f'f2: {f2(2)}')
print(f'f3: {f3(2)}')
print(f'f4: {f4(f2,2)}')
print(f'f4: {f4(f3,2)}')
```

Scope and Lifetime - namespaces

- Scope of a Variable (determines visibility):
    - The scope of a variable defines the region of the program where the variable is visible and accessible. In Python, there are four main types of variable scopes:
    - Local Scope: A variable declared inside a function or block is considered to have a local scope. It can only be accessed within that function or block and is not visible to other functions or blocks.
    - Enclosing Scope: When a function is defined inside another function, the inner function has access to variables from the outer (enclosing) function's scope.
    - Global Scope: A variable declared outside any function or block has a global scope. It can be accessed from any part of the program, including inside functions.
    - Built-in Scope: Python has some built-in names that are available in all scopes without explicitly importing them. For example, names like print, len, etc., are part of the built-in scope.
- Lifetime of a Variable (determines survival):
    - The lifetime of a variable is the period during which the variable exists in memory. It starts when the variable is created and ends when it is deleted. The lifetime of a variable depends on its scope:
    - Local Variables: The lifetime of a local variable starts when the function is called and ends when the function exits. After the function returns, the local variables are destroyed, and their memory is released.
    - Global Variables: The lifetime of a global variable is the entire duration of the program. It remains in memory as long as the program is running or until explicitly deleted.
    - Enclosing and Built-in Variables: Their lifetime is the same as that of the functions or modules that enclose them or until explicitly deleted.

```
x =10
def f1(a):
    x = 20
print(x)
def f2(x):
    x = 30
print(x)

def f3(x=40):
    pass
print(x)

def f4(x=25):
    return x + 3
print(f4(x))

print(f4())
print(f4(3.3))
```

```
x = 10  # global scope
print(x) # print function is built-in scope
while(True):
    x = 20
    print(x)
    for x in range(3):
        print(x)
    for p in range(3):
        print(p)
    break
print(f'p: {p}')

print(x)

while(True):
    a11 = 9.3
    print(a11)
    break
print(a11)
```

```
def f5():
    a9 = 10.1
    print(a9)
    def f6():
        print(a9) # a9 available from enclosing scope
    f6()

f5()

#print(a9) # here a9 is not defined as the a9 is in local scope of f5 function
```

```
f1(a,'a'); a


#the re-assignment in local scope is not reflected in parent scope
#this is different than modifying the passed object through its method such as append
#as we saw above
def f1(l,x):
    l = [1,2]


def f1_mod(l,x):
    #l = [1,2]
    l.append(x)
    print(l)
    return l

a1 = [2,3,'a']
a = f1_mod(a1,1);print(a);print(a1)


f1(a,1); a
```

More on Parameters and Arguments

- local scope created - an identifier in local scope of the function has no relation with the identifier of the same name in the calling scope
- default return is None
- parameter passing <=> assignment statement between formal and actual parameters (alias creation)
- return value also follows assignment statement sematics (so no replication of objects)
- re-assigning a new value to a formal parameter within a function body does not affect the actual parameter (re-assignment breaks the alias)
- default parameters (enabling more than one possible calling signature) - the caller can provide a variable number of parameters when calling
- if a default parameter value is present for one parameter, it must be present for all further parameters - so my_func(x,y=10,z) cannot be a function signature
- positional argument - the actual parameters are matched in sequence to the formal parameters - order or sequence is IMPORTANT
- keyword argument - specified by explicity assigning the actual value to the formal parameter name while calling e.g. when function signature is my_func(x=5,y=6,z=2) and we call it by using my_func(z=2) then x and y are assigned the default values and c is assigned the explicitly passed value

Points to Note in Functions

The default values are evaluated at the point of function definition in the defining scope

```
i = 5
def f(arg=i):
    print(arg)

i = 6
f( )
```

The default value is evaluated only once – study the difference between the following two function definitions:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

———————————————

```
def f(a, L=None):
    if L is None:
        L = [ ]
    L.append(a)
    return L
```

Keyword Arguments

- In a function call, keyword arguments must follow positional arguments
- All the keyword arguments passed must match one of the arguments accepted by the function and their order is not important, this also includes non-optional arguments
- No argument may receive a value more than once


```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")

# valid statements
parrot(1000)                                    # 1 positional argument
parrot(voltage=1000)                            # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')       # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)       # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')   # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword


# invalid statements - uncomment one by one to check response
#parrot()                     # required argument missing
#parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
#parrot(110, voltage=220)     # duplicate value for the same argument
#parrot(actor='John Cleese')  # unknown keyword argument
```

Lambda functions

- short, one line, in-lined functions
- unnamed or anonymous
- used and discarded immediately - cannot be revoked later on
- lambda param: body


```
l1 = [(3,1),(2,3),(8,-1),(4,2)]
m = max(l1, key=lambda x: x[1])
print(m)
```

Function Doc & Function Annotations

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass


print(my_function.__doc__)
```

- Function annotations are completely optional metadata information about the types used by user-defined functions - they are not enforced

```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

f('spam')


def f(b: int, i: int) -> list:
    return [b,i]

f(3.14,2.1)
```

Built-in functions

abs(x)	Return the absolute value of a number.
all(iterable)	Return True if bool(e) is True for each element e.
any(iterable)	Return True if bool(e) is True for at least one element e.
chr(integer)	Return a one-character string with the given Unicode code point.
divmod(x, y)	Return (x // y, x % y) as tuple, if x and y are integers.
hash(obj)	Return an integer hash value for the object
id(obj)	Return the unique integer serving as an “identity” for the object.
input(prompt)	Return a string from standard input; the prompt is optional.
isinstance(obj, cls)	Determine if obj is an instance of the class (or a subclass).
iter(iterable)	Return a new iterator object for the parameter
len(iterable)	Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)	Return an iterator yielding the result of function calls f(e1, e2, ...) for respective elements e1 ∈ iter1, e2 ∈
max(iterable)	Return the largest element of the given iteration.
max(a, b, c, ...)	Return the largest of the arguments.
min(iterable)	Return the smallest element of the given iteration.
min(a, b, c, ...)	Return the smallest of the arguments.
next(iterator)	Return the next element reported by the iterator
open(filename, mode)	Open a file with the given name and access mode.
ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers) equivalent to x ** y.
pow(x, y, z)	Return the value (x^y mod z) as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values 0, 1, . . . , stop − 1.
range(start, stop)	Construct an iteration of values start, start + 1, . . . , stop − 1.
range(start, stop, step)	Constructaniterationofvaluesstart,start+step,start+2 step,...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

Built-in functions

ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers); equivalent to x ** y.
pow(x, y, z)	Return the value (x^y mod z) as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values 0, 1, . . . , stop − 1.
range(start, stop)	Construct an iteration of values start, start + 1, . . . , stop − 1.
range(start, stop, step)	Constructaniterationofvaluesstart,start+step,start+2 step,...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

Useful String Functions

- useful methods of string objects - join( ), strip( ), format( ), capitalize( ), endswith( ), count( ), index( ), etc.

```
s1 = 'this is a string'

s1.capitalize( )
s1.endswith('ring')
s1.index('is',3)
s1.count('is')
s1.isnumeric( )
s1.strip('th')
s1.split(' ')

':'.join(s1)

s1 = 'this is { }\'s class on { }'.format('john','wednesday')

name = 'john'; day = 'wednesday'
s1 = f'this is {name}\'s class on {day}'
```

Conditional Expression

- expr1 if condition else expr2
- the output of a conditional expression can be directly assigned to a variable
- a conditional expression can directly be a parameter in a function call

Packing and Unpacking of Sequences

```
x = 3,5,2,1
print(f'{x[0]} {x[3]}')

q,r = divmod(7,3)
print(f'{q} {r}')

for m,n in [(1,4),(5,6),(2,2)]:
    print(f'{m} - {n}')

d = {'a': 'apricot', 'b': 'banana', 1: 'guava'}

for k,v in d.items():
    print(f'{k} :: {v}')

a,b,c = 'pradeep', 'rajeev','sandip'
print(f'{a} {b}')

a,b = b,a
print(f'{a} {b}')
```

Iterators

- Iterator and iterable objects enable iteration protocol
- used by for loops, in membership test, map, etc.
- Iterables can provide iterators which is like a cursor and helps to go over a sequence one after the other
- syntax: for element in iterable
- iterable: list, tuple, set, str (on characters), dict (on keys), file (on lines)
- user defined iterables (advanced topic when discussing Object Oriented Python)
- iteration mechanism: iterator (object for controlling iteration), iterable (produces iterator object - iter(obj))

```
l1 = [1,6,4,3]
i = iter(l1)
try:
    while True:
        print(next(i))
except StopIteration:
    print('iteration over')

for e in l1:
    print(e)

j = iter(l1)
print(next(j))
print(next(j))
l1[2]=l1[2]*5
print(next(j))
```

Special Iterators

- lazy evaluation - supply when demanded - saves memory, unused values not evaluated
- range - list(range(10)) or in for loop
- dict.keys( ), dict.values( ), dict.items( )

Generators

- generators are iterators
- generators are functions
- lazy evaluation
- yield instead of return
- generator keeps generating as many values as the number of yield statement it contains when invoked inside a loop or called directly using next syntax
- CAREFUL: after the generator has been exhausted dont expect anything from it again

Generator examples

```
def mygen3():
    yield 'a'
    yield 'b'
    yield 'c'
    yield 'c'
for i in mygen3():
    print(i)

y = mygen3()
print(next(y))
print(next(y))
list(y)

def mygen4(j):
    for i in range(j):
        yield i*2

for k in mygen4(4):
    print(k)
```

Generator Expressions vs Generator Functions

```
# generator expression similar to list comprehension
gen1 = (x**2 for x in range(10) if x % 2==0)

#generator objects support iterator protocol and can be used over for loop
for i in gen1:
    print(i)

print('\n')

#rebuild the generator object to again fetch from it
gen1 = (x**2 for x in range(10) if x % 2==0)
```

More examples on Generators

- generator_demo.py

Comprehension

- Series in Series out - create a new series from an existing series
- applies to list [ ], set{ }, dict{k:v } and generator ( )
- [expr for value in iterable if condition]

```
[ k*2 for k in range(1, n+1) ]   # list comprehension
{ k*2 for k in range(1, n+1) }   # set comprehension
( k*2 for k in range(1, n+1) )   # generator comprehension
{ k : k*2 for k in range(1, n+1) }   # dictionary comprehension
```

Exception Handling - I

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions
- Exceptions (errors) - raised / thrown under error conditions and special conditions
- Exceptions are caught in order to "gracefully" handle raised or thrown exception - if unhandled program will ultimately terminate / crash

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.attrib1, if obj has no member named attrib1
EOFError	Raised if “end of file” reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Exception Handling - II

- Raising exception
- try - except - finally - one "try" (essential), zero or more "except" (i.e. optional), one optional else, one optional final "finally"


```
def my_add(x,y):
    return x+y

try:
    x = int(1)
    #x = int('a')  # will throw a ValueError
    #my_add(2,'a')  # will throw a TypeError
    #'a' + 10 # will throw a TypeError
    #print(qq)  # will throw a NameError if qq has not been defined in local, outer or global scope
    raise Exception('param1','param2') # raising an exception programmatically with optional parameters
except ValueError:
    print("Oops!  That was no valid number.  Try again...")
except (TypeError, NameError) as e:
    print("It is either a TypeError or NameError ::", e)
    print(f'caught {type(e)}')
except Exception as e:
    print(e.args[1])
else:
    print("everything went fine (no exception occured). you may like to do wrap up activities here such as closing open files, etc.")
finally:
    print("this will always execute w/ or w/o exception occuring")
```

Print formatting

- formatted string literals (f-strings):

```
year = 2016
event = 'Referendum'
f'Results of the {year} {event}'
```

- str.format( )

```
print('We are the {} who say "{}!"'.format('knights', 'Ni'))
print('{1} and {0}'.format('spam', 'eggs'))
```

- Old formatting method (% operator)

```
import math
print('The value of pi is approximately %5.3f.' % math.pi)
print('%(language)s has %(number)03d quote types.' % {'language': "Python", "number": 2})
```

Import and Modules

- modules - collection of values, functions and classes
- modules are imported
- modules have their own namespace
- script vs module
- from math import pi, sqrt
- from math import pi as p
- from math import *
- import math
- math.pi
- math.sqrt
- my_module.py
- import my_module
- main module

- A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

Module Name	Description
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Module Search path

- if searching for module my_module
- first search among built-in modules with same name - sys.builtin_module_names
- builtin modules - all modules that are compiled into this Python interpreter
- next search for a file my_module.py in a list of directories given by sys.path
- sys.path - initialised to directory containing the current script, PYTHONPATH, installation dependent default (site-packages directory)
- built-in function dir( ) is used to find out which names a module defines
- dir (module_name)
- dir( ) #list the names current defined by the programmer
- import builtins
- dir(builtins) # see next page

Builtin functions - from dir(builtins)

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'ModuleNotFoundError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',
 'RecursionError',
 'ReferenceError',
 'ResourceWarning',
 'RuntimeError',
 'RuntimeWarning',
 'StopAsyncIteration',
 'StopIteration',
 'SyntaxError',
''SyntaxWarning',
 'SystemError',
 'SystemExit',
 'TabError',
 'TimeoutError',
 'True',
 'TypeError',
 'UnboundLocalError',
 'UnicodeDecodeError',
 'UnicodeEncodeError',
 'UnicodeError',
 'UnicodeTranslateError',
 'UnicodeWarning',
 'UserWarning',
 'ValueError',
 'Warning',
 'ZeroDivisionError',
 '__IPYTHON__',
 '__build_class__',
 '__debug__',
 '__doc__',
 '__import__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'execfile',
'filter',
'float',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'runfile',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'vars',
'zip']
```

Important Modules

- sys - sys.argv, sys.path, sys.version, sys.module
- os - getcwd( ), listdir( ), mkdir( ), path.join( ), path.exists( ), path.isdir( ), name, sep
- re - regular expressions in python (to be covered later)

Input and File Handling

```
name = input('What is your name')
name

age = int(input('What is your age'))
age

my_file = open('my_python_module.py')
my_file.read()

my_file = open('my_python_module.py')
for line in my_file:
    print(line)

my_file = open('my_python_module.py')
list(my_file)

my_file = open('my_python_module.py')
my_file.readlines()

my_file = open('my_python_module.py')
l = my_file.readline()
while l:
    print(l)
    l = my_file.readline()

# best
with open('my_python_module.py',mode='r', encoding='utf-8') as f:
    lines = f.readlines()
    lines = [line.strip() for line in lines]
    print(lines)
```

json module

- load and dump json data from objects, files
- serialisation and deserialisation

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

```
import json
d1 = {'a': [(1,2),(3.4)],'b':[(2,3),(4,4)]}
s1 = json.dumps(d1)
s2 = json.loads(s1)
s2.keys( )
isinstance(s2,dict)
```

```
f = open('my_json_dump.txt','w')
l1 = ['a1', {'a2': ('t1', None, 5, False,3.2)}]
json.dump(l1,f)
f.close( )
```

```
f = open('my_json_dump.txt')
l1 = json.load(f)
f.close( )
```

Future sessions

- Object orientation - classes and objects
- Custom iterables, iterators, exceptions
- Container classes
- Collections module
- Decorators and decorator classes
- Designing modules and packages
- Regular expressions