Python Basics

Invoking Python

Short programs can be executed directly on command line:

```
$ python -c "import datetime; print(datetime.datetime.now())"
2015-08-11 18:42:38.929678
```

More commonly, programs are stored in files and executed from command line. In this example, the above python commands are stored in file *now.py* and the output is printed:

```
$ python now.py
2015-08-11 18:42:38.929678
```

For learning and exploring the language, Python interactive shell is ideal:

```
$ python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2015, 8, 11, 18, 42, 38, 929678)
>>> print(datetime.datetime.now())
2015-08-11 18:42:38.929678
>>>
```

A more powerful shell popular among programmers is IPython. For editing Python programs, there are plenty of open source editors and IDEs available for download.

When writing Python programs, it is common practice among programmers to follow the PEP8 Style Guide.

Third-party packages can be downloaded and installed on top of default Python installation. These can be found at PyPi. The easiest way to install an external package is to use the *pip* tool. For example, to install the package named Sphinx, one would type the following:

```
$ pip install Sphinx
```

Data Types

Python is a dynamically typed language. Types are bound to values at run time, not variables at compile time. Basic types include the following:

- `int`: Integer type.
- `float`: Float type.
- `bytes`: Typically used to contain the encoded form of any text containing Unicode characters.
- `str`: String type.
- `list`: A container of multiple items where items can be of different types. Other languages may refer to such a type as array.
- `tuple`: Similar to a list expect that once assigned it cannot be changed. Typically used to pass parameters and return values in and out of functions.
- `dict`: A dictionary or an associative array of key-value pairs. Keys have to be unique. There is no order in how items are stored.
- `set`: Similar to list except that it contains unique items. Like dict, items are not stored in any particular order.

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> b = 2.0
>>> type(b)
<class 'float'>
>>> c = 'Hello'
>>> type(c)
<class 'str'>
>>> d = chr(234)
>>> d
'ê'
>>> e = d.encode('utf-8')
>>> e
b'\xc3\xaa'
>>> type(e)
<class 'bytes'>
>>> f = [a, b, c, d, e]
>>> type(f)
<class 'list'>
>>> g = (a, b, c, d, e)
>>> g
(1, 2.0, 'Hello', 'ê', b'\xc3\xaa')
>>> type(g)
<class 'tuple'>
>>> h = {'int':a, 'float':b, 'str':c}
>>> h
{'float': 2.0, 'int': 1, 'str': 'Hello'}
>>> type(h)
<class 'dict'>
>>> L = [1, 2, 3, 2, 3]
>>> set(L)
{1, 2, 3}
>>>
```

Some data types are immutable: once assigned their values cannot be changed. If immutable types are completely reassigned, a new object is created to replace the old. Mutable types are data types that can be modified after initial assignment. These include list, set and dict.

Data types str, bytes, list and tuple are sometimes called as *sequences* because items have order. Because of this order, these data types can be indexed with an integer to retrieve a particular item. Types set and dict are not sequences since the items in them have no order. Types set and dict are sometimes called as *collections*.

For sequences, in addition to indexing a single item, a range of items can be obtained by using slicing. For example, given a list L, L[1:10:2] implies a slice of alternate items from L[1] to L[9]. Slicing is defined by L[start:end:step]. If start and end are omitted, the entire list is processed. If step is omitted, every item is processed. Negative step value implies processing in reverse order from the end of the sequence to its start.

```
>>> L = ['a', 'b', 'c', 'd']
>>> L[0]
'a'
>>> L[3]
'd'
>>> s = 'Hello'
>>> s[0]
'H'
>>> s[-1]
'o'
>>> L[1:3]
['b', 'c']
>>> s[1:3]
'el'
>>> s[::2]
'Hlo'
>>> s[::-1]
'olleH'
>>>
```

When defining strings, use of double quote is same single quote, unlike some other languages. Strings with newlines in them are usually defined using triple quotes.

```
>>> long_line = """This is a sentence that
... runs into multiple lines."""
>>> print(long_line)
This is a line that
runs into multiple lines
>>>
```

Using converters, it's possible to convert from one data type to another:

```
>>> a = 'A'
>>> ord(a)
65
>>> chr(65)
'A'
>>> hex(65)
'0x41'
>>> oct(65)
'0o101'
>>> bin(65)
```

```
'0b1000001'
>>> float(65)
65.0
>>> str(65)
'65'
>>> int('65')
65
>>>
```

Basic Operators

The well-known arithmetic operators common in many other languages are supported:

```
>>> print (3 + 4) # addition
7
>>> print (3 - 4) # subtraction
-1
>>> print (3 * 4) # multiplication
12
>>> print (3 / 4) # division
0.75
>>> print(3 % 2) # modulo
1
>>> print(3 ** 4) # exponential
81
>>> print(3 // 4) # floor division
0
>>>
```

The above can be combined with assignment operator to yield the following short hands:

```
>>> a = 3
>>> a += 4
>>> a
7
>>> a -= 4
>>> a
3
>>> a *= 4
>>> a
12
>>> a /= 4
>>> a
3.0
>>> a %= 2
>>> a
1.0
>>> a = 8
>>> a **= 0.5
>>> a
2.8284271247461903
>>> a //= 2
>>> a
1.0
```

```
>>>
```

The following comparison operators are supported and they return the `bool` data type:

```
>>> 3 > 4
False
>>> 3 < 4
True
>>> 3 >= 4
False
>>> 3 <= 4
True
>>> 3 == 4
False
>>> 3 != 4
True
>>> type(False)
<class 'bool'>
>>>
```

Note that == does value comparison but the `is` operator does an object identity comparison:

```
>>> s1
'hello'
>>> s2
'olleh'
>>> s1 == reversed(s2)
False
>>> s1 == ''.join(reversed(s2))
True
>>> s1 == s2[::-1]
True
>>> s1 is s2[::-1]
False
>>> id(s1)
40337568
>>> id(s2[::-1])
40337624
>>>
```

Following logical operators (`and`, `or`, `not`) are supported:

```
>>> True and False
False
>>> True or False
True
>>> True and not False
True
>>>
```

Membership of items in sequences and collections can be tested using the `in` operator:

```
>>> L = [1, 2, 4, 6, 8, 10]
>>> 8 in L
```

```
True
>>> 3 in L
False
>>> 4 not in L
False
>>> D = {'int': 1, 'float': 2.0, 'str': 'hello'}
>>> 'str' in D
True
>>> 'bool' in D
False
>>>
```

Bitwise operators are also supported: `&  |  ^  ~  <<  >>`

String Methods

The following examples illustrate some commonly used methods:

```
>>> S = "I Love Python Programming!"
>>> S.lower()
'i love python programming!'
>>> S.upper()
'I LOVE PYTHON PROGRAMMING!'
>>> S.count('o')
3
>>> S.replace('Love','Like')
'I Like Python Programming!'
>>> S.find('g')
17
>>> S.rfind('g')
24
>>> S.find('x')
-1
>>> S.strip('!')
'I Love Python Programming'
>>> L = ['a', 'b', 'c']
>>> ':'.join(L)
'a:b:c'
>>>
```

Methods `split` and `join` form a useful pair to convert between strings and lists. For example, the following code allows us to remove leading and trailing spaces, as well as replace multiple spaces with a single space:

```
>>> S = "   I     Love   Python  Programming!    "
>>> ' '.join(S.split())
'I Love Python Programming!'
>>>
```

One can overwrite a string but cannot modify an individual character in-place since strings are immutable:

```
>>> S = "Hello World!"
>>> S = S.upper()
>>> S
'HELLO WORLD!'
>>> S[0] = 'Y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Repetition and concatenation of strings can sometimes be useful:

```
>>> 'ab' + 'cd'
'abcd'
>>> 'ab'*3
'ababab'
>>>
```

The recommended way to format string is to use the `format` method:

```
>>> a = 8
>>> print("Square root of {:d} is {:.6f}".format(a, a**0.5))
Square root of 8 is 2.828427
>>>
```

Often web pages will contain non-ASCII characters. It is important to know how to handle them using `str.encode` and `bytes.decode` methods:

```
>>> book = 'shrīmad bhagavad gītā'
>>> book.encode()
b'shr\xc4\xabmad bhagavad g\xc4\xabt\xc4\x81'
>>> print('Chars: {:d}; Bytes: {:d}'.format(len(book), len(book.encode())))
Chars: 21; Bytes: 24
>>> rs = '\u20b9'
>>> rs
'□'
>>> type(rs)
<class 'str'>
>>> rs.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u20b9' in position
0: ordinal not in range(128)
>>> rs.encode('utf-8')
b'\xe2\x82\xb9'
>>> rs.encode('utf-8').decode()
'□'
>>>
```

List Methods

The following examples illustrate some commonly used methods:

```
>>> L
['a', 'b', 'c']
>>> L.reverse()
>>> L
['c', 'b', 'a']
>>> L.append('d')
>>> L
['c', 'b', 'a', 'd']
>>> L.sort()
>>> L
['a', 'b', 'c', 'd']
>>> L.extend(['e', 'f'])
>>> L
['a', 'b', 'c', 'd', 'e', 'f']
>>> L.pop()
'f'
>>> L
['a', 'b', 'c', 'd', 'e']
>>> L.remove('c')
>>> L
['a', 'b', 'd', 'e']
>>> L.count('a')
1
>>> L.count('f')
0
>>> M = L.copy()
>>> M
['a', 'b', 'd', 'e']
>>> L == M
True
>>> L is M
False
>>> M.insert(2, 'z')
>>> M
['a', 'b', 'z', 'd', 'e']
>>>
```

An empty list evaluates to False upon a Boolean operation. The same can be said of an empty string.

```
>>> L = []
>>> bool(L)
False
>>> L = ['']
>>> bool(L)
True
>>> bool(L[0])
False
>>> L.clear()
>>> L
[]
>>>
```

Set Methods

Unlike lists, sets contain unique items and items are stored without any order.

```
>>> S = {1, 2, 1, 2, 3, 4}
>>> S
{1, 2, 3, 4}
>>> S.add(5)
>>> S
{1, 2, 3, 4, 5}
>>> S.add(4)
>>> S
{1, 2, 3, 4, 5}
>>> S.update({2, 9, 4})
>>> S
{1, 2, 3, 4, 5, 9}
>>> len(S)
6
>>> S.pop()
1
>>> S
{2, 3, 4, 5, 9}
>>> S.remove(5)
>>> S
{2, 3, 4, 9}
>>> S.clear()
>>> S
set()
>>>
```

Sets in python are easy to handle. Basic set theory operations can be implemented easily:

```
>>> s = {1, 2, 3}
>>> t = {3, 4, 5}
>>> s & t
{3}
>>> s | t
{1, 2, 3, 4, 5}
>>> s ^ t
{1, 2, 4, 5}
>>> s - t
{1, 2}
>>> t - s
{4, 5}
>>>
```

Tuple Methods

Unlike list, tuple data type has only a couple of methods: `count` and `index`. This is because tuples are immutable.

```
>>> T = (1, 2, 3, 1, 1, 4)
>>> T[0]
1
>>> T[0::2]
(1, 3, 1)
```

```
>>> T.count(1)
3
>>> T.count(4)
1
>>> T.index(3)
2
>>> T.index(4)
5
>>>
```

Dictionary Methods

The following examples illustrate some commonly used methods:

```
>>> D = {'int': 1, 'float': 2.0, 'str': 'hello'}
>>> D.keys()
dict_keys(['float', 'int', 'str'])
>>> D.values()
dict_values([2.0, 1, 'hello'])
>>> D.items()
dict_items([('float', 2.0), ('int', 1), ('str', 'hello')])
>>> D.get('int')
1
>>> D.get('int',2)
1
>>> D.get('lang','Python')
'Python'
>>> L = ['a', 'b', 'c']
>>> dict.fromkeys(L)
{'c': None, 'b': None, 'a': None}
>>> dict.fromkeys(L,0)
{'c': 0, 'b': 0, 'a': 0}
>>> D
{'float': 2.0, 'int': 1, 'str': 'hello'}
>>> D.popitem()
('float', 2.0)
>>> D
{'int': 1, 'str': 'hello'}
>>> D['str']
'hello'
>>> E = D.copy()
>>> E
{'int': 1, 'str': 'hello'}
>>> E['year']  = 2015
>>> E
{'year': 2015, 'int': 1, 'str': 'hello'}
>>> E.pop('int')
1
>>> E
{'year': 2015, 'str': 'hello'}
>>> E.clear()
>>> E
{}
>>>
```

Control Statements

Program execution flow can be controlled using the `if-elif-else` statement. Code belonging to a particular branch is indented. Nested conditions imply nested indentations. There is no *switch-case* statement found in some other languages.

```
L = ['a', 'b', 'c']
if L:
    if len(L)<5:
        print("List is very small.")
    elif len(L)<10:
        print("List is small.")
    else:
        print("List is big.")
else:
    print("List is empty.")
```

Loops and Comprehensions

Python has a `while-else` loop but this is rarely used by programmers who prefer to use `for-else` loop. Like some other languages, `break` and `continue` statements are possible inside loops. The following code does a prime number test for integers between 2 and 9. Note that the else statement is executed if the inner loop runs completely without breaking.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

While for loops are powerful, the more elegant way to program in Python is to use *comprehensions*. Consider the case of printing the cube of numbers from 0 to 9. Both approaches are shown below:

```
# Using traditional for loop
cubes = []
for n in range(10):
    cubes.append(n**3)
print(cubes)

# Using list comprehension
print([n**3 for n in range(10)])
```

A list comprehension has essentially three parts: `[`*`expression loop-iteration+ filter`*`]`. The filter, which is usually an `if` statement, is optional. Loop iteration can be a single loop or a nesting of many loops. Thus, the first few *Pythagorean Triplets* (x, y, z) satisfying the condition $x^2+y^2=z^2$ can be easily computed as follows:

```
>>> [(x, y, z) for z in range(1,100) for y in range(1, z) for x in range(1,
y) if x*x + y*y == z*z]
[(3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15), (8, 15, 17), (12, 16, 20),
(15, 20, 25), (7, 24, 25), (10, 24, 26), (20, 21, 29), (18, 24, 30), (16, 30,
34), (21, 28, 35), (12, 35, 37), (15, 36, 39), (24, 32, 40), (9, 40, 41),
(27, 36, 45), (30, 40, 50), (14, 48, 50), (24, 45, 51), (20, 48, 52), (28,
45, 53), (33, 44, 55), (40, 42, 58), (36, 48, 60), (11, 60, 61), (39, 52,
65), (33, 56, 65), (25, 60, 65), (16, 63, 65), (32, 60, 68), (42, 56, 70),
(48, 55, 73), (24, 70, 74), (45, 60, 75), (21, 72, 75), (30, 72, 78), (48,
64, 80), (18, 80, 82), (51, 68, 85), (40, 75, 85), (36, 77, 85), (13, 84,
85), (60, 63, 87), (39, 80, 89), (54, 72, 90), (35, 84, 91), (57, 76, 95),
(65, 72, 97)]
>>>
```

The following code illustrates dictionary comprehension that uses the word as key and word length as value. In addition, the comprehension does filtering to include only those words that contain the letter 'i'.

```
>>> languages = ['Hindi', 'Kannada', 'Urdu', 'Tamil', 'Sanskrit', 'Punjabi']
>>> {x:len(x) for x in languages if 'i' in x}
{'Hindi': 5, 'Tamil': 5, 'Sanskrit': 8, 'Punjabi': 7}
>>>
```

Built-in Functions

The following examples illustrate some commonly used built-in functions:

```
>>> L = ['a', 'b', 'c']
>>> len(L)
3
>>> tuple(enumerate(L))
((0, 'a'), (1, 'b'), (2, 'c'))
>>> M
['a', 'z', 'e']
>>> max(M)
'z'
>>> list(zip(L,M))
[('a', 'a'), ('b', 'z'), ('c', 'e')]
>>> T = tuple(range(0,10,2))
>>> T
(0, 2, 4, 6, 8)
>>> all(T)
False
>>> any(T)
True
>>> list(map(bool, T))
[False, True, True, True, True]
>>> list(filter(bool, T))
[2, 4, 6, 8]
>>> help(list.reverse)
Help on method_descriptor:

reverse(...)
    L.reverse() -- reverse *IN PLACE*
```

```
>>>
```

With reference to the above, `enumerate` returns 2-item tuples with first item as index and second item as value. If we wish to combine two sequences item wise, then `zip` is a handy function. To create sequences or sets of integers, `range` is a useful function. The syntax of range is similar to that of slicing operator with start, stop and step as arguments. Function `all` returns True if each item of the container evaluates to True. On the contrary, `any` returns True if at least one item evaluates to True. Function `map` allows us to apply a certain function on each item of a container. Likewise, `filter` can return a sub-container based on certain filtering criteria. Since comprehensions are powerful, map and filter operations are not usually used.

A full list of built-in functions is available.

Functions

Like control statements, indentation specifies where the function body starts and ends. As the following code shows, arguments to a function can be positional arguments (dummy, k) or keyword arguments (name, place). Arguments could also be specified to take default arguments (name, place). Unused arguments (dummy) will not result in any error but it's probably a good idea to remove them.

In terms of variables, those assigned inside a function have local scope (j, k). Global immutables can be used inside a function for read access but not for write access, unless `global` keyword is used (i). Global mutables can be modified inside a function (count).

```
def sample_func(dummy, k, name='Python',place='India'):
    'Print name and place introduction.'
    global i
    i += 1
    # ERROR: j += 1
    j = -1
    if name in count:
        count[name] += 1
        j = count[name]
        k += 1
    print('[{:d},{:d},{:d}]: I am {:s} from {:s}'.format(i, j, k, name,
place))
    return j

i, j, k = 0, 99, 123
count = dict.fromkeys(['Python','Anaconda'], 0)
print(sample_func)
print('Return:',sample_func(None, k))
print('Return:',sample_func(None, k, 'Anaconda'))
print('Return:',sample_func(None, k, place='Africa'))
print('Return:',sample_func(None, k, 'Anaconda', 'Africa'))
print('Return:',sample_func(None, k, 'Cobra', 'India'))
print(count, k)
```

Running the above code will give the following output:

```
<function sample_func at 0xb71bf974>
[1,1,124]: I am Python from India
Return: 1
[2,1,124]: I am Anaconda from India
Return: 1
[3,2,124]: I am Python from Africa
Return: 2
[4,2,124]: I am Anaconda from Africa
Return: 2
[5,-1,123]: I am Cobra from India
Return: -1
{'Python': 2, 'Anaconda': 2} 123
```

When passing tuples in and out of functions, *packing* and *unpacking* are commonly done. Packing means that individual arguments are combined to form a tuple and then processed. Packing is useful when function can take a variable number of arguments. Unpacking is the reverse and is often used to assign items of a tuple to individual variables to improve code readability.

```
# Unpacking
def add_couple(num1, num2):
    """Add two numbers together"""
    return num1 + num2

numbers = (22, 20)
print(add_couple(*numbers))

# Packing
def add_all(*nums):
    """Add any number of numbers together"""
    if nums:
        return nums[0] + add_all(*nums[1:]) # unpacking
    return 0

print(add_all(18, 5, 19))
print(add_all(*range(100))) # unpacking
```

Modules and Packages

Python files of extension **.py* are called modules. Packages are directories that typically contain one or more modules. A package must contain a file named *__init__.py*. This file is normally used to do package level initializations or export only specific modules or functions. Both modules and packages help in organizing one's code and enabling code reuse.

In the following example, four functions are imported from the *math* module. Note the use of packing and unpacking.

```
from math import atan2, sqrt, sin, cos
def cartesian_to_polar(x, y):
    return (sqrt(x*x+y*y), atan2(y,x))
```

```python
def polar_to_cartesian(r, theta):
    return (r*cos(theta), r*sin(theta))

if __name__ == '__main__':
    print(cartesian_to_polar(3,4))
    print(cartesian_to_polar(*(3,4)))
    print(polar_to_cartesian(*cartesian_to_polar(3,4)))
```

Suppose we save the above code in file *polar.py*. Thus, we have created our own module named *polar*. We can execute the module directly and the code within the `__name__ == '__main__'` condition will run. We can alternatively import the module from another Python script:

```python
import polar
print(polar.cartesian_to_polar(5,6))
```

File I/O

The recommended way to work with files is to use the `with` statement. This is because once the scope of the statement is exited, the file is automatically closed.

```python
import os
path = 'examples.py'
try:
    with open(path, 'r') as f:
        print(f.read())
except IOError:
    print('Unable to read')
```

The use of packages `sys` and `os` can simplify the programming task of file I/O. The following shows how one can recursively print out the number of files and size in bytes starting from any given directory:

```python
import sys
import os
for path, dirs, files in os.walk(sys.argv[1]):
    tot_size = sum([os.path.getsize(os.path.join(path, name)) for name in 
files])
    print('{:s}: {:d} files, {:d} bytes'.format(path, len(files), tot_size))
    if '.git' in dirs:
        dirs.remove('.git')
```

Following is an example execution of the above code:

```
$ python examples.py ~/workspace/python
/home/tejas/workspace/python: 25 files, 91453 bytes
/home/tejas/workspace/python/__pycache__: 8 files, 29529 bytes
/home/tejas/workspace/python/data: 10001 files, 703401 bytes
```

Exceptions

Exceptions occur when the program does something unexpected at run time. The following code illustrates some exceptions defined in core Python:

```
>>> L
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'L' is not defined
>>> L = [1, 4, 10]
>>> L[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> L.remove(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> L[0] + "This is a string"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

When working with exceptions, it is worth remembering the following:

- User-defined exceptions must derive from `Exception`.
- Use `finally` to do cleanup or release resources.
- Keep exception classes simple, just to pass/process information.
- Use exceptions to handle unexpected events.
- Don't use exceptions for control flow. This is as bad as using "goto" statements.
- Don't catch exceptions to hide bugs. Worst code is one that catches `BaseException`.
- Log exceptions with complete error message.
- Catch and re-raise exceptions to avoid exposing implementation details.
- Always document what exceptions can be raised by a function/method.

The following example illustrates a few different ways in which exceptions can be used. Try running it and interpret the results.

```
import sys
import traceback
def process_list(lst):
    print(lst)
    print("***print_stack: ", file=sys.stderr)
    traceback.print_stack()
    try:
        if not lst:
            tb = sys.exc_info()[2]
            raise Exception("List is empty",'EMPTY').with_traceback(tb)
        lst[0]/lst[1]
    except (TypeError,ZeroDivisionError):
        etype, evalue, etb = sys.exc_info()
        print("***print_exception: ", file=sys.stderr)
        traceback.print_exception(*sys.exc_info()) # aka 
traceback.print_exc()
        print("***print_tb: ", file=sys.stderr)
        traceback.print_tb(etb)
    except IndexError as e:
        print('IndexError', e) # scope of e limited to except
    else:
        print('Everything ok')
    finally:
        print('Completed list processing\n\n')
    #print('Variable out of scope? {:s}'.format(repr(e)))

try:
    process_list(['a', 'b', 'c'])
    process_list([0, 0, 'c'])
    process_list([0])
    process_list([1, 2, 3])
    process_list([])
except Exception as e:
    print('Outside exception: {:s}'.format(repr(e)))
    traceback.print_exc()
```

Intermediate / Advanced Python

- Object-oriented Programming
- Debugging and Testing Code
- Profiling and Optimizing Code
- Concurrent Programming
- Regular Expressions
- Practice with Modules
- Working with Data Files and Formats
- Interfacing with Databases
- Functional Programming
- Networking at Application Level
- Documenting and Distributing Code

- Introduction

In this chapter, we will not cover the concepts of object-oriented programming (OOP). The focus will be on understanding OOP from the perspective of Python implementation and usage.

Everything in Python is an object. This includes basic data types such as integers and strings. Even functions are objects. Python has been designed from the ground up with object-oriented programming (OOP) in mind. In large projects involving dozens of files and hundreds of functions, OOP becomes necessary. Even in smaller projects, OOP can improve code maintenance and reduce bugs.

The use of modules and packages allows for modular implementation. The use of classes and objects takes this approach to the next step of code reuse and dynamic control. Multiple objects can be created based on the same class.

A Simple Class

The following class defines a user account with two *attributes*: email and password. Two objects are created or instantiated: u1 and u2. We can also say that u1 and u2 are *instances* or *instance objects* of the class. Thus, many users can be created from the same class. In simple terms, the class is like a mould or template from which objects can be created.

```
class User:
    """User account with login information."""

    def __init__(self, email, password):
        self.email = email
        self.password = password

u1 = User('tejaspdmn@iedf.in', 'abcd1234')
u2 = User('johnsmith@gmail.com', 'johnny12')
print(u1, u1.email, u1.password)
print(u2, u2.email, u2.password)
```

We can add more functionality to the User class as shown below. An instance object can perform attribute references. Attributes are either data or methods. Functions defined inside a class when accessed via an instance are called *methods*. Method `__init__` is a special one, also known as the *constructor*. It is called when the object is instantiated. As a first argument, it is passed the object. This argument is typically named `self`.

Data attributes hold the state or configuration of the object. Methods implement the behaviour of the object. Thus, email and password are data. Methods set_name, get_name, log_access and access_count implement behaviour. Note that methods can call other methods and data attributes of the object via `self`. Attributes that start with double underscore will be treated as private to the class. Hence, they will not be accessible from outside the class. This is Python's way of implementing *data hiding*.

```
from datetime import datetime
```

```python
class User:
    """User account with login information."""

    def __init__(self, email, password):
        self.email = email
        self.password = password
        self.__login_ts = []

    def set_name(self, firstname='', surname=''):
        self.firstname = firstname
        self.surname = surname

    def get_name(self):
        name = ''
        if hasattr(self, 'firstname'):
                name = self.firstname
        if hasattr(self, 'surname'):
                name += ' ' + self.surname
        return name.strip()

    def log_access(self):
        self.__login_ts.append(datetime.now())

    def access_count(self):
        return len(self.__login_ts)

u2 = User('johnsmith@gmail.com', 'johnny12')
u2.set_name('John', 'Smith')
u2.log_access()
u2.log_access()
u2.log_access()
print(u2, u2.email, u2.password, u2.get_name(), u2.access_count())
print(u2.__login_ts) # raises AttributeError
```

Runtime Modifications

Due to the dynamic nature of Python--many things happen at run time rather than at compile time--it is possible to add attributes to an object after it has been instantiated. This can be done to affect all instances of the class or a specific instance. When methods are added dynamically in this manner, it is sometimes called *monkey patching*. This is usually not a good practice but sometimes useful in creating test stubs or in patching broken code on a live application while the team works for a long term solution.

```python
class Point:
    pass

p1 = Point()
print(p1.__dict__)
print(dir(p1))

# add data attributes
p1.x, p1.y, p1.z = 1, 3, 5
print(p1.__dict__)
print(dir(p1))
print(p1.x, p1.y, p1.z)
```

```
# add methods: sometimes called "monkey patching"
def from_origin(self):
    return (self.x**2 + self.y**2 + self.z**2)**0.5
def walklen(self):
    return self.x + self.y + self.z

# add method to the class
Point.from_origin = from_origin
print("Function bound to instance p1: %s" % p1.from_origin)
print("Distance from origin:", p1.from_origin())

# add method to a specific instance: failure case
p2 = Point()
p2.x, p2.y, p2.z = 1, 3, 5
try:
    p2.walklen = walklen
    print("Unbound function: %s" % p2.walklen)
    print("Walk length:", p2.walklen())
except TypeError as e:
    print("Cannot call unbound function: %s" % e)
del(p2.walklen)

# add method to a specific instance: using types.MethodType
import types
try:
    p2.walklen = types.MethodType(walklen, p2)
    print("Function bound to instance p2: %s" % p2.walklen)
    print("Walk length:", p2.walklen())
except TypeError as e:
    print("Cannot call unbound function: %s" % e)
del(p2.walklen)

# add method to a specific instance: using functools.partial
import functools
try:
    p2.walklen = functools.partial(walklen, p2)
    print("Function bound to instance p2: %s" % p2.walklen)
    print("Walk length:", p2.walklen())
except TypeError as e:
    print("Cannot call unbound function: %s" % e)
del(p2.walklen)
```

Functions as Objects

Although functions are not instantiated from classes, they too are objects. The following example shows how the function cube can be passed into another function as an argument:

```
def cube(x):
    return x*x*x

def add(f, a, b):
    return f(a)+f(b)

n = (3,4)
print('{0}**3 + {1}**3 = {result}'.format(*n, result=add(cube,*n)))
print(cube, add)
```

Because functions are also objects, it's quite possible to have nested definitions of functions. Thus, a function can return another function. This powerful feature is used to implement *decorators*. A decorator performs a transformation on a function, thus enhancing or decorating the function's behaviour. The beauty of decorators is that this can be done without changing the function's implementation or the way it is called. In the following example, a decorator is used to deliberately delay processing of the function.

```
from time import sleep

def sleeper(func):
    def wrapper(*args, **kwargs):
        sleep(1)
        return func(*args, **kwargs)
    return wrapper

@sleeper
def printer(num):
    print('This is a number:', num)

for x in range(1,6):
    printer(x)
```

Since a function can be nested inside another function, this is used to implement *closure*. Closure means that the nested function is bound to a particular configuration and returned by the outer function. The following example shows how two functions two_pow and three_pow are "closed" instances of the inner function:

```
def generate_power(exponent):
    def nth_power(power):
        return exponent**power
    return nth_power

two_pow = generate_power(2)
three_pow = generate_power(3)
print(two_pow(7)) # 2**7 = 128
print(three_pow(5)) # 3**5 = 243

# Inspect closure values
import inspect
print(inspect.getclosurevars(two_pow))
print(inspect.getclosurevars(three_pow))
```

Since closures can be implemented as decorators, the decorator approach is preferred by programmers:

```
def generate_power(exponent):
    def wrapper(func):
        def inner(*args, **kwargs):
            result = func(*args, **kwargs)
            return exponent**result
        return inner
    return wrapper

@generate_power(2)
def two_pow(n):
    return n

@generate_power(3)
def three_pow(n):
    return n

print(two_pow(7))
print(three_pow(5))
```

The following code illustrates closure based on our earlier example on sleeper. It is implemented as a function inside a function inside a function. A closure is really a decorator that accepts arguments.

```
from time import sleep

def sleeper(delay):
    def outer(func):
        def wrapper(*args, **kwargs):
            sleep(delay)
            return func(*args, **kwargs)
        return wrapper
    return outer

@sleeper(2)
def printer(num):
    print('This is a number:', num)

for x in range(1,6):
    printer(x)
```

In the above example, the decorator is a function. Taking an object-oriented approach, we can implement it as a class. Magic method `__init__` is used to initialize the arguments. Magic method `__call__` returns the decorated function.

```
from time import sleep

class sleeper:
    def __init__(self, delay):
        self.delay = delay

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            sleep(self.delay)
            return func(*args, **kwargs)
        return wrapper


@sleeper(3)
def printer(num):
    print('This is a number:', num)
```

```
for x in range(1,6):
    printer(x)
```

Class Attributes

Sometimes all instances of a given class may need to share the same data attribute or method. This is where class attributes become useful. Class methods takes the class object as its first argument. Yes, classes are treated as objects that are created when they are defined. Class attributes are therefore bound to class objects. Class methods can be created using the decorator `@classmethod`.

Where a method uses no attributes of the class, it could be defined as a static method. This could be done using the `@staticmethod` decorator. Alternatively, such a method could be defined as a module level function outside the class. Thus, static methods are probably useful if one wants to implement behaviours private to the class.

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    @staticmethod
    def sound():
        print('Woof! Woof!')

    @classmethod
    def describe(cls):
        print('I am a {0}.'.format(cls.kind))

    def __init__(self, name):
        self.name = name  # instance variable unique to each instance

    def greet(self):
       print("Hi! I'm", self.name)

d1 = Dog('Sparky')
d2 = Dog('Spotty')
Dog.sound()
d1.sound()
Dog.describe()
d1.describe()
print(Dog.kind, d1.kind, d1.name)
print(Dog.kind, d2.kind, d2.name)
print(d1.sound, d1.describe, d1.__init__)
print(d2.sound, d2.describe, d2.__init__)

import time
print(id(d1.sound), id(d1.describe), id(d1.greet))
time.sleep(1)
print(id(d1.sound), id(d1.describe), id(d1.greet))
time.sleep(1)
print(id(d2.sound), id(d2.describe), id(d2.greet))
time.sleep(1)
print(id(d2.sound), id(d2.describe), id(d2.greet))
```

```
d1.greet()
Dog.greet(Dog('Barky'))
Dog.greet() # raises a TypeError
```

In the above example, we can see that static methods and class methods/attributes can be called via the instances or the class names. The following output is produced. Note that static methods are treated like function objects. Class methods are bound to the class. Other methods are bound to instance objects. Methods such as greet() must be called via a valid instance object.

```
Woof! Woof!
Woof! Woof!
I am a canine.
I am a canine.
canine canine Sparky
canine canine Spotty
<function Dog.sound at 0xb71b7df4> <bound method type.describe of <class
'__main__.Dog'>> <bound method Dog.__init__ of <__main__.Dog object at
0xb71b872c>>
Hi! I'm Sparky
Hi! I'm Barky
Traceback (most recent call last):
  File "examples.py", line 30, in <module>
    Dog.greet() # raises a TypeError
TypeError: greet() missing 1 required positional argument: 'self'
```

Inheritance and Composition

Inheritance is an OOP concept that defines relationships between classes. Designers often think in terms of classes and relationships. This helps them to identify different parts of the problem. It helps them to break down a complex problem into simpler parts. It makes the design more modular. For example, we can define BankAccount as a generic class. SavingsAccount and CurrentAccount are different types of BankAccount. SavingsAccount is a type of BankAccount; hence this is an *is-a* relationship. In this case, BankAccount is called the *base class*. SavingsAccount is called the *derived class*.

During implementation, inheritance enables code reuse since much of the behaviour of base class can be used by the derived class. It is also possible for the derived class to customize its own behaviour. This is usually done by redefining a method already present in the base class. Thus, method in a derived class *overrides* one in the base class.

What about a bank transaction? This too could be defined as a class named Transaction. Clearly, bank account contain multiple transactions. In this case, rather than *is-a* relationship, we call this a *has-a* relationship. Formally, this is known as *composition* because a bank account is composed of transactions.

```
class BankAccount:
    def __init__(self, branch, acc_num):
        self.branch = branch
        self.acc_num = acc_num
        self.__balance = 0

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, value):
        self.__balance = value

class SavingsAccount(BankAccount):
        pass

class CurrentAccount(BankAccount):
    def __init__(self, min_balance, od_limit, *args, **kwargs):
        self.min_balance = min_balance
        self.od_limit = od_limit
        super().__init__(*args, **kwargs)

sa1 = SavingsAccount('JPNagar', '001287498')
print('Savings Account {0} @ {1}:'.format(sa1.acc_num, sa1.branch))
print(sa1.balance)
sa1.balance = 1000
print(sa1.balance)
sa1.balance += 500
print(sa1.balance)

ca1 = CurrentAccount(1000, 5000, 'Domlur', '44238904')
print('Current Account {0} @ {1}:'.format(ca1.acc_num, ca1.branch))
print(ca1.balance, ca1.min_balance, ca1.od_limit)
ca1.balance = 10000
print(ca1.balance, ca1.min_balance, ca1.od_limit)
```

In the above example, SavingsAccount has nothing to modify. It simply uses the behaviour and data attributes of BankAccount. On the other hand, CurrentAccount adds two new attributes and overrides the `__init__` method. Note how CurrentAccount is implemented so that it processes data meant for itself and passes on the rest to BankAccount. The use of `*args` and `*kwargs`, ensures that other classes downstream may derive from CurrentAccount without difficulty because all classes in the inheritance hierarchy will have matching function signatures. This is known as *cooperative inheritance*.

Another thing to note is the use of decorators `@property` and `@balance.setter`. This allows us to protect `__balance` attribute while retaining it's access by the name of `balance`. The advantage of this approach is that we can keep the interface fixed while changing the internal implementation at any time.

The use of composition can be useful at times when inheritance is too cumbersome. Composition allows an object to do selective processing and delegate the rest to the member object. For this purpose, the magic method `__getattr__` is called whenever an attribute that doesn't belong a class is accessed. The class can then delegate the processing to the member object. The following example is an illustration of composition although in this simple case inheritance is perhaps more suitable:

```
class BankAccount:
    def __init__(self, branch, acc_num):
        self.branch = branch
        self.acc_num = acc_num
        self.__balance = 0

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, value):
        self.__balance = value

class SavingsAccount():
    def __init__(self, number, status):
        self.__account = BankAccount(number, status)

    def __getattr__(self, attr):
        return getattr(self.__account, attr)


sa1 = SavingsAccount('JPNagar', '001287498')
print('Savings Account {0} @ {1}:'.format(sa1.acc_num, sa1.branch))
print(sa1.balance)
sa1.balance = 1000
print(sa1.balance)
sa1.balance += 500
print(sa1.balance)
```

Magic Methods

These are special methods that are named with leading and trailing double underscores. They define behaviours "under the hood." We have already seen the `__init__` method that is responsible for initializing an object when it is created. There are also `__new__` and `__del__` that are called when object is created or deleted respectively.

For numeric types, it is common to have arithmetic and comparison operators. These are implemented as magic methods. For example, it is not possible to compare two `complex` numbers. However, the following example shows how we can derive a new class named zcomplex and add comparison behaviours to it. Strictly speaking, to sort a list we need only `__lt__`, though we have defined many more magic methods for other comparisons on zcomplex.

```
from random import randint
class zcomplex(complex):
    def __init__(self, obj):
        self.obj = obj
    def __lt__(self, other):
        return self.obj.real < other.obj.real
    def __gt__(self, other):
        return self.obj.real > other.obj.real
    def __eq__(self, other):
        return self.obj.real == other.obj.real
    def __le__(self, other):
        return self.obj.real <= other.obj.real
    def __ge__(self, other):
        return self.obj.real >= other.obj.real
    def __ne__(self, other):
        return self.obj.real != other.obj.real

print('\n\nSorting on zcomplex:')
z = [zcomplex(randint(-100,100)+randint(-100,100)*1j) for _ in range(5)]
print(z)
print(sorted(z))

print('\n\nSorting on complex:')
c = [randint(-100,100)+randint(-100,100)*1j for _ in range(5)]
print(c)
print(sorted(c))
```

Other useful magic methods to keep in mind are `__iter__` and `__next__` for adding iterator behaviour to class; `__contains__` to test membership; `__enter__` and `__exit__` to create context managers; `__str__`, `__repr__` and `__format__` to obtain printable versions of the object. It is worth studying some of these.

Attributes that have leading double underscores but not trailing underscores are considered as private to the object. The Python interpreter mangles their names so that they become inaccessible from outside. Attributes with a single leading underscore are a "weak" implementation of private access. They can be accessed from outside but they will not be imported from modules when doing `from` *module_name* `import *`. The following example illustrates private access:

```
class Person():
    def __init__(self, name, age, address):
        self.name = name
        self._age = age
        self.__address = address

    def show_hidden(self):
       print(self.__address)

class Employee(Person):
    def __init__(self, employer, salary, *args, **kwargs):
        self._employer = employer
        self.__salary = salary
        super().__init__(*args, **kwargs)

    def show_hidden(self):
       print(self.__salary)
       super().show_hidden()

    def __salary__(self):
       self.__show_salary()

    def __show_salary(self):
       print(self.__salary)


p1 = Person('Roopa', 25, 'Chennai')
print(p1.name, p1._age)
```

```
p1.show_hidden() # indirectly access hidden data attributes

e1 = Employee('Intel', 1200000, 'John', 34, 'Bangalore')
print(e1._employer, e1.name, e1._age)
e1.show_hidden() # indirectly access hidden data attributes
e1.__salary__() # can access magic methods
e1.__show_salary() # cannot access hidden methods
```

Method Resolution Order

Python allows multiple inheritance. In other words, a class can have multiple base classes. The order in which the base classes are used matters. Multiple base classes will have a common ancestry since all classes are ultimately coming from `object` class at the root. To avoid conflicts, there is a strict order in which base class methods are called. This is called *Method Resolution Order (MRO)*.

The following example illustrates MRO. D derives from both B and C in that order. When D calls `super()` to access the base class object, this call is resolved to B. When B calls `super()`, it resolves to C, which is the next class in the inheritance hierarchy of D. Note that B's call to `super()` obtains C although B is not derived from C in any way. This illustrates the power of inheritance in Python and the usefulness of MRO. If C did not have an implementation of the particular method (who_am_i), then B's call of `super()` would resolve to A. Finally, there is only one `object` in the inheritance hierarchy of D, even though A, B and C are all derived from `object`. Likewise, D has only one A object. This example illustrates the *diamond pattern*, which occurs because multiple interitance paths lead to the same base class, such as, D-B-A and D-C-A both leading to class A.

```
class A(object):
    def who_am_i(self):
        print("I am a A")

class B(A):
    def who_am_i(self):
        print("I am a B")
        print("My parent says: ", end='')
        super().who_am_i()

class C(A):
    def who_am_i(self):
        print("I am a C")
        print("My parent says: ", end='')
        super().who_am_i()

class D(B,C):
    def who_am_i(self):
        print("I am a D")
        print("My parent says: ", end='')
        super().who_am_i()

d1 = D()
d1.who_am_i()
print(D.mro())
```

Running the above would print the following:

```
I am a D
My parent says: I am a B
My parent says: I am a C
My parent says: I am a A
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>]
```

In the trivial case where there is no diamond pattern, then the MRO is obvious. For example, if C is derived from another class E(object) in the above example, then the MRO for D would be D-B-A-C-E-object.

A practical example of using multiple inheritance is illustrated using the `collections` module. `Counter` is a class that counts occurrences of characters and returns a dict. `OrderedDict` is a dictionary that preserves the order in which key-value pairs are added. By combining these two, we can therefore form an OrderedCounter that does counting as well as preserves the order.

```
from collections import Counter, OrderedDict

class OrderedCounter(Counter, OrderedDict):
     """Counter that remembers the order elements are first seen"""
     def __repr__(self):
         return '%s(%r)' % (self.__class__.__name__,
                            OrderedDict(self))
     def __reduce__(self):
         return self.__class__, (OrderedDict(self),)

oc = OrderedCounter('abracadabramagic')
print(oc)
print(Counter.mro())
print(OrderedCounter.mro())
```

The above example is due to Raymond Hettinger, one of the Python's key contributors. It is clear that Counter derives from dict. OrderedCounter on the other hand resolves to OrderedDict first and dict later. Executing the above would print the following:

```
OrderedCounter(OrderedDict([('a', 6), ('b', 2), ('r', 2), ('c', 2), ('d', 1),
('m', 1), ('g', 1), ('i', 1)]))
[<class 'collections.Counter'>, <class 'dict'>, <class 'object'>]
[<class '__main__.OrderedCounter'>, <class 'collections.Counter'>, <class
'collections.OrderedDict'>, <class 'dict'>, <class 'object'>]
```

Class-related Functions

The following examples illustrate some common functions:

```
class A(object):
    def who_am_i(self):
        print("I am a A")

class B(A):
    def who_am_i(self):
        print("I am a B")

b1 = B()
b1.who_am_i()

if not hasattr(b1, 'name'):
    print("b1 has no name")
    b1.name = "Tejas"

if hasattr(b1, 'name'):
    print("Name:", b1.name)

print("Email:",getattr(b1, 'email', None))
setattr(b1, 'email', 'tejas@gmail.com')
print("Email:",getattr(b1, 'email', None))
delattr(b1, 'email')
if not hasattr(b1, 'email'):
    print("b1 has no email")

if isinstance(b1, B):
    print("I am an instance of B")
if isinstance(b1, A):
    print("I am an instance of A")
if not isinstance(b1, dict):
    print("I am NOT an instance of dict")
if issubclass(B, A):
    print("I am B and I am derived from A")
if issubclass(int, object):
    print("I am int and I am derived from object")
```

Executing the above code will give the following:

```
I am a B
b1 has no name
Name: Tejas
Email: None
Email: b@gmail.com
b1 has no email
I am an instance of B
I am an instance of A
I am NOT an instance of dict
I am B and I am derived from A
I am int and I am derived from object
```

Abstract Base Classes

These provide an interface (methods) without an implementation. ABCs should not be instantiated. Instead they specify an interface that derived classes should implement. Derived classes can then be instantiated into objects.

The usual way to create and use ABCs is via the module `abc`. This allows us to catch missing implementations when objects are instantiated, instead of catching errors later when particular methods are called.

```python
import abc

class Shape(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def area(self):
        """Returns the shape's area."""

class Square(Shape):
    def __init__(self, side):
        self.side = side

class Rectangle(Shape):
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def area(self):
        return self.length * self.breadth

try:
    s0 = Shape()
except TypeError:
       print("Cannot instantiate Shape")

r1 = Rectangle(3, 5)
print(r1.area())

try:
    s1 = Square(3)
except TypeError:
       print("Cannot instantiate Square")
```

ABCs are also useful to tell the interpreter that a certain class conforms to the interface of another even though it is not derived from the latter. Thus, anyone calling `isinstance()` or `issubclass()` on this class will get expected behaviour. The following example shows how one mimics Vector class as the base class of the built-in `complex` class.

```python
from abc import ABCMeta

class Vector(metaclass=ABCMeta):
    pass

Vector.register(complex)
print(issubclass(complex, Vector))print(complex.mro())
```

- Conclusion

  Classes and objects are the cornerstone of Python's power and elegance. Concepts of OOP as relevant to Python are summarized below:

    - Encapsulation: Data and functions related to manipulating that data are contained within a class. Encapsulation points to a *has-a* relationship, such as bank accounts have money.
    - Data Hiding: Attributes can be kept private by prefixing double underscore to their names.
    - Inheritance: This is where one object inherits certain attributes of another (or others). Methods in the derived class can also override those of the base class. Inheritance points to a *is-a* relationship, such as a savings account is a type of bank account.
    - Polymorphism: This is where a single interface can give rise to different run time behaviour depending on who is calling it. In Python, it is MRO that determines run time behaviour.

What's Next?

Without repeating what's already been covered in this brief tutorial, the following study links will improve your understanding of Python:

1. The Python Standard Library
   There are of course lots of core modules to learn but these may be useful starting points: `os`, `os.path`, `glob`, `shutil`, `sys`, `string`, `datetime`, `copy`, `zipfile`, `csv`, `json`, `pickle`, `re`, `math`, `random`, `argparse`, `operator`, `functools`, `itertools`, `inspect`, `threading`.
2. Transforming Code into Beautiful, Idiomatic Python (Video)
   This is an essential video that every beginner should watch.
3. PEP 20 -- The Zen of Python
4. Common Pitfalls
    - Common Python Pitfalls (Udacity)
    - Buggy Python Code: The 10 Most Common Mistakes That Python Developers Make
5. Python Namespace and Scope
   There are subtle differences with the way namescapes and scoping work in other languages. It would be useful to understanding these well before doing serious Python programming.
6. Iterators, Generators and Decorators
7. Improve Your Python: Understanding Unit Testing
   Often unit testing is ignored in many introductory courses, suggesting that perhaps it's an optional topic. Good programmers always exercise their code with matching unit tests. Unit testing is becoming more important these days with the rise of *Test-Driven Development (TDD)* and *DevOps*.
8. Improve Your Python: Python Classes and Object Oriented Programming
   Your knowledge of Python will remain incomplete unless you understand objects, classes, inheritance and other concepts related to object-oriented programming.
9. Duck Typing in Python