A BRIEF INTRODUCTION TO PYTHON

(With content based on the book “A Byte of Python”, by Swaroop [3])

Modules

- Although in a somewhat simplistic way, we can say:
  - *to reuse a set of lines of code in the same program, we use **functions**;*
  - *to reuse a set of functions in different programs, we use **modules**;*
- A module is essentially a set of functions that can be integrated into any program.
  - *It can consist of a file with **.py** extension (or **.pyc**), containing functions and variables in Python,*
  - *but it can also be written and compiled, for example in C, and then used within Python code.*

Modules

- A program to use the functionalities of a module must first import it.
  - *This is also what should be done when we want to use the modules in the standard Python library.*
- When the module is imported, its contents run

Example (save as `module_using_sys.py`):

```
import sys

print('The command line arguments are:')
for i in sys.argv:
    print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

```
$ python module_using_sys.py we are arguments    # each arg is separated by white space
```

- The import sys command, searches for the sys module
  - *and knows where it is given that it is an internal module;*
  - *if it wasn't, it would be searched in one of the folders of the variable sys.path.*
- The variable sys.argv is a list of strings containing the arguments passed in the program invocation, the first of which is the name of the program itself running

Output:

```
The command line arguments are:
module_using_sys.py
we
are
arguments


The PYTHONPATH is ['/tmp/py',
# many entries here, not shown here
'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
```

Compiled modules for bytecode

- The import of modules can prove to be a somewhat heavy operation
  - *One way to make this operation less heavy is to convert the module into a .pyc file*
  - *A .pyc file contains bytecode, an intermediate code that results from the compilation of the original Python code,*
    - being of faster execution
    - and independent on the platform.

Statement from... Import

- If we want to import not the entire module, but only one variable or function (an object) from that module, we must use the ***from***.. ***Import***
  - *For example, to directly import the **argv** variable into the program (to avoid using the sys qualifier again and again, the declaration must be used*

```
from sys import argv
```

  - *Or, to import the **sqrt** function of the 'math' module, it is written*

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

The attribute __name__

- A module always has a name
  - *this name can be queried from within the module itself, during its execution, accessing the attribute* `__name__`
  - *As can be seen in the example, it is possible, through this attribute, to see if a module is running directly or by import*

Example (save as `module_using_name.py`):

```
if __name__ == '__main__':
    print('This program is being run by itself')
else:
    print('I am being imported from another module')
```

Output:

```
$ python module_using_name.py
This program is being run by itself

$ python
>>> import module_using_name
I am being imported from another module
>>>
```

How to create a module

- As a Python program is itself a module, we can say that we have been creating several modules for a long time

  (the modules must be placed in the same folder of the program that imports them, or else in one of the folders contained in sys.path)

Example (save as `mymodule.py`):

```
def say_hi():
    print('Hi, this is mymodule speaking.')

__version__ = '0.1'
```

Another module (save as `mymodule_demo.py`)

```
import mymodule

mymodule.say_hi()
print('Version', mymodule.__version__)
```

utilising the `from..import` syntax

```
from mymodule import say_hi, __version__

say_hi()
print('Version', __version__)
```

Output:

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

The function dir()

- The *dir()* function returns the list of names defined by an object
  - *If this object is a module, the list includes the names of the functions, classes, and variables defined in that module.*
- If no argument is used in the *dir()* function, the list of names of the current module is returned

```
$ python
>>> import sys

# get names of attributes in sys module
>>> dir(sys)
['__displayhook__', '__doc__',
'argv', 'builtin_module_names',
'version', 'version_info']
# only few entries shown here

# get names of attributes for current module
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__', 'sys']

# create a new variable 'a'
>>> a = 5

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys', 'a']

# delete/remove a name
>>> del a

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

Packages

- So far, some hierarchical organization of Python programs has been noticeable:
  - *the local variables are within functions;*
  - *the global variables and functions lies within modules;*
- And how to organize the modules?
  - *this is where the packages come into play…*
  - *packages are no more than module folders, with a special file:* `__init__.py`
    - it is through this file that Python realizes that the folder is special (especially because it contains modules)

Subpackages

- Each Package can contain other Packages, which we designate *subpackages*.

- The folder structure that is illustrated in the example, is intended to represent the creation of a package called 'world' with the subpackages 'asia' and 'africa', and these in turn with the supackages 'india' and 'madagascar', each containing a module, foo.py and bar.py, respectively.

```
- <some folder present in the sys.path>/
    - world/
        - __init__.py
        - asia/
            - __init__.py
            - india/
                - __init__.py
                - foo.py
        - africa/
            - __init__.py
            - madagascar/
                - __init__.py
                - bar.py
```

Data Structures

Python provides us with 4 data structures (collections) that make life a lot easier for us...

- List – heterogeneous and ordered collection of objects (ordered in the sense that each element occupies a well-determined position in the structure, not depending on that position of the order ratio of the values themselves)
  - *in the specification of a list, elements must be placed within a pair of straight parentheses: list=[2, 5, -3]*
  - *is a variable structure, since it allows us to remove and add new elements*
  - *dictionaries are instances of the list class*

- Tuple – also a heterogeneous and orderly collection of objects but with fewer functionalities
  - *in the specification of a tuple, the elements are placed, optionally, within a pair of curved parentheses: tuple1=(2, 5, -3); tuple2=1, 2, 3;*
  - *is an immutable structure, as it does not allow changes to its content,*
  - *commonly used when it is intended to ensure that the collected elements are not changed*
  - *tuples are instances of the tuple class*

Data Structures

- Dictionary – key/value pair collection
  - *in the specification of a dictionary, 'key:value' pairs are placed within a pair of keyrooms: dicionario={'Ana':18, 'Rui':15, 'Rita':20}*
  - *keys have to be immutable objects*
  - *already the values can be immutable or not*
  - *dictionaries are instances of the dict class*
- Set – unordered heterogeneous collection without the possibility of repetitions
  - *set(: c1=set([2,5,3]), c2=set(((2,5,3)), c1=set("253")*
  - *sets are instances of the set class*
- Sequence – it is a more conceptual and comprehensive type of collection, which presents as main characteristics the fact of allowing access to its members through the operations 'in' and 'not in' (membership tests), and through indexing.
  - *Lists, tuples, and strings are sequences*

Example of using a list

```
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print('I have', len(shoplist), 'items to purchase.')

print('These items are:', end=' ')
for item in shoplist:
    print(item, end=' ')

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My shopping list is now', shoplist)

print('I will sort my list now')
shoplist.sort()
print('Sorted shopping list is', shoplist)
print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

Do not forget that the list itself (of objects) is an object

- therefore has a set of features (methods) that facilitate their handling of the
  - *To know all your methods, just do: help(list)*

```
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print('I have', len(shoplist), 'items to purchase.')

print('These items are:', end=' ')
for item in shoplist:
    print(item, end=' ')

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My shopping list is now', shoplist)

print('I will sort my list now')
shoplist.sort()
print('Sorted shopping list is', shoplist)
print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

Output

```
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

Use of tuples

```
zoo = ('python', 'elephant', 'penguin')
print('Number of animals in the zoo is', len(zoo))

new_zoo = 'monkey', 'camel', zoo    # parentheses not required but are a good idea
print('Number of cages in the new zoo is', len(new_zoo))
print('All animals in new zoo are', new_zoo)
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is',
      len(new_zoo)-1+len(new_zoo[2]))
```

Output:

```
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python', 'elephant', 'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

Use of tuples - return of 2 values

- One of the interesting applications of tuples is in creating functions that return not one, but several results.
  - *See an example function that returns two values*

```
>>> def get_error_details():
...     return (2, 'details')
...
>>> errnum, errstr = get_error_details()
>>> errnum
2
>>> errstr
'details'
```

- When we can use a tuple to the left of an assignment, this will allow, for example, the exchange of the value of two variables to be done in a very simple way

```
>>> a = 5; b = 8
>>> a, b
(5, 8)
>>> a, b = b, a
>>> a, b
(8, 5)
```

Using a dictionary

The method items() of the dictionary ab returns a list of tuples, each containing a key/value pair

- *In the example, each pair is assigned to the name and address variables using the for.. In*

```
ab = {   # 'ab' is short for 'a'ddress'b'ook
    'Swaroop': 'swaroop@swaroopch.com',
    'Larry': 'larry@wall.org',
    'Matsumoto': 'matz@ruby-lang.org',
    'Spammer': 'spammer@hotmail.com'
}

print("Swaroop's address is", ab['Swaroop'])

# Deleting a key-value pair
del ab['Spammer']

print('\nThere are {} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():
    print('Contact {} at {}'.format(name, address))

# Adding a key-value pair
ab['Guido'] = 'guido@python.org'

if 'Guido' in ab:
    print("\nGuido's address is", ab['Guido'])
```

```
ab = {   # 'ab' is short for 'a'ddress'b'ook
    'Swaroop': 'swaroop@swaroopch.com',
    'Larry': 'larry@wall.org',
    'Matsumoto': 'matz@ruby-lang.org',
    'Spammer': 'spammer@hotmail.com'
}

print("Swaroop's address is", ab['Swaroop'])

# Deleting a key-value pair
del ab['Spammer']

print('\nThere are {} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():
    print('Contact {} at {}'.format(name, address))

# Adding a key-value pair
ab['Guido'] = 'guido@python.org'

if 'Guido' in ab:
    print("\nGuido's address is", ab['Guido'])
```

Output

```
Swaroop's address is swaroop@swaroopch.com

There are 3 contacts in the address-book

Contact Swaroop at swaroop@swaroopch.com
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org

Guido's address is guido@python.org
```

Sets

- Sets are unordered collections of unrepeated objects.
  - *With these data structures it is possible to perform typical cluster operations, such as:*
    - check if one set is a subset of another,
    - and find the intersection between two sets.

```
>>> bri = set(['brazil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brazil', 'india'}
```

Use of sequences

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'

# Indexing or 'Subscription' operation #
print('Item 0 is', shoplist[0])
print('Item 1 is', shoplist[1])
print('Item 2 is', shoplist[2])
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])

# Slicing on a list #
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])

# Slicing on a string #
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

Output:

```
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Character 0 is s

Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']

characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

References

- As with what happens, for example, in Java and C#, when you assign an object directly to a variable, that variable will only contain a reference to the same object, not a copy of it.
  - *Sometimes, to counteract this behavior (i.e., to duplicate the object) in the assignment of sequences, you use the 'slicing' operation in the sequence assignment (see next example)*

References

```
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
# mylist is just another name pointing to the same object!
mylist = shoplist

# I purchased the first item, so I remove it from the list
del shoplist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that both shoplist and mylist both print
# the same list without the 'apple' confirming that
# they point to the same object

print('Copy by making a full slice')
# Make a copy by doing a full slice
mylist = shoplist[:]
# Remove first item
del mylist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that now the two lists are different
```

Output:

```
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

Still about strings

- When it comes to objects, strings have a wide variety of methods that help us do almost everything we need...
  - *are instances of the str class*
  - *for a complete listing of your methods, use* `help(str)`

```
# This is a string object
name = 'Swaroop'

if name.startswith('Swa'):
    print('Yes, the string starts with "Swa"')

if 'a' in name:
    print('Yes, it contains the string "a"')

if name.find('war') != -1:
    print('Yes, it contains the string "war"')

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

Output:

```
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

Reading user data

```
def reverse(text):
    return text[::-1]

def is_palindrome(text):
    return text == reverse(text)

something = input("Enter text: ")
if is_palindrome(something):
    print("Yes, it is a palindrome")
else:
    print("No, it is not a palindrome")
```

Output:

```
Enter text: sir
No, it is not a palindrome

Enter text: madam
Yes, it is a palindrome

Enter text: racecar
Yes, it is a palindrome
```

solve exercise #3 and #4

from the book of exercises