Python crash course

Functional python programming

with **Jongbin Jung**

All material publicly available here (https://github.com/jongbinjung/mse231-workshop)

2. python programming - 2 of 3

Now that we're quite familiar with some of python's basic concepts, let's touch some 'advanced' topics.

Writing functions

Let's create a function to count the number of vowels in a given string

In [1]:
```
def count_vowels(s):
    """Count the number of vowels in a string."""
    vowels = 'aeiouAEIOU'
    nvowels = [s.count(v) for v in vowels]  # count the number of eac
h vowel in s
    return sum(nvowels)  # return the sum of elements in nvowel

# use the new function
count_vowels('Eels are delicious animals')
```

Out[1]: 12

- the `def` keyword declares a function **def**inition, followed by a function name and the parenthesized list of formal parameters
- the statements that form the body of the function start at the next line, and must be indented
- the first statement of the function body can optionally be a string, also known as the docstring (https://docs.python.org/2/tutorial/controlflow.html#tut-docstrings)
- many tools (such as `spyder`) use the docstring to give users meaningful information - so help yourself, make a habit of writing meaningful docstrings
- functions that don't finish with a `return` statement return `None` (a special `python` object for "Nothing")

Functions can also return a tuple of values. For example, let's modify our `count_vowels` function to return the number of vowel along with a `list` specifying the number of each vowel.

In [2]:
```
def count_vowels(s):
    """
    Count the number of vowels in a string.

    returns: number of vowels, list containing number of appearance f
or each vowel
    """
    vowels = 'aeiouAEIOU'
    nvowels = [s.count(v) for v in vowels]  # count the number of eac
h vowel in s
    return sum(nvowels), list(zip(vowels, nvowels))  # return the sum
 and a zipped list

count_vowels('Eels are delicious animals')
```

Out[2]:
```
(12,
 [('a', 3),
  ('e', 3),
  ('i', 3),
  ('o', 1),
  ('u', 1),
  ('A', 0),
  ('E', 1),
  ('I', 0),
  ('O', 0),
  ('U', 0)])
```

A returned tuple can also be 'unpacked' into multiple variables.

In [3]:
```
total_count, individual_count = count_vowels('Eels are delicious anim
als')
print 'Found total', total_count, 'vowels, each vowel as follows:'
print individual_count
```

```
Found total 12 vowels, each vowel as follows:
[('a', 3), ('e', 3), ('i', 3), ('o', 1), ('u', 1), ('A', 0), ('E',
 1), ('I', 0), ('O', 0), ('U', 0)]
```

Functions with optional arguments

Let's further enhance the `count_vowels` function by letting the user specify

- which vowels to count ('aeiouAEIOU' by default)
- whether to return a single sum or a tuple of the sum and list (single sum by default)

This can be achieved by specifying default values in the function declaration.

In [4]:
```
def count_vowels(s, vowels = 'aeiouAEIOU', returnAll = False):
    """
    Count the number of vowels in a string.

    Arguments:
    s -- the string to count vowels from
    vowels -- string of characters that should be considered a vowel
 (default: aeiouAEIOU)
    returnAll -- boolean indicating whether to return just the sum of
 vowels (default: False)
                 or a tuple of the sum of vowels and a list of occere
nce for each character

    returns: number of vowels[ , list containing number of appearance
 for each vowel]
    """
    nvowels = [s.count(v) for v in vowels]  # count the number of eac
h vowel in s
    if returnAll:
        return sum(nvowels), list(zip(vowels, nvowels))  # return the
 sum and a zipped list
    else:
        return sum(nvowels)  # return just the sum

count_vowels('Eels are delicious animals')
```

Out[4]:
```
12
```

In [5]:
```
count_vowels('Eels are delicious animals', vowels = 'aeiou')  # no ca
ps
```

Out[5]:
```
11
```

In [6]:
```
count_vowels('Eels are delicious animals', returnAll = True)  # give
 me EVERYTHING
```

Out[6]:
```
(12,
 [('a', 3),
  ('e', 3),
  ('i', 3),
  ('o', 1),
  ('u', 1),
  ('A', 0),
  ('E', 1),
  ('I', 0),
  ('O', 0),
  ('U', 0)])
```

Be careful with having mutable defaults, though. Default values of a function's argument are shared between subsequent calls, and this might cause problems if you're manipulating the argument's value within the function. For example,

In [7]:
```
def fun(n, stuff=[]):
    """Illustrating issues with mutable defaults."""
    stuff.append(n)
    return stuff

print fun(1)  # stuff is empty by default
print fun(2)  # stuff was manipulated, and is now [1] from the previo
us call!
print fun(3)  # even worse, stuff is now [1, 2] !!!
```

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

This behavior isn't necessarily a problem, and it might even make sense in some contexts. However, it's definitely worth keeping in mind to avoid being surprised. If you want to prevent such behavior, one simple work-around is to set the default to `None`, and check if it is indeed `None`, before assigning the 'true' default, such as:

In [8]:
```
def fun(n, stuff=None):
    """Fix for mutable defaults."""
    if stuff is None:
        stuff = []
    stuff.append(n)
    return stuff

print fun(1)  # unspecified argument stuff is None, then set to []
print fun(2)  # unspecified argument stuff is None, then set to []
print fun(3)  # unspecified argument stuff is None, then set to []
print fun(3, [1,2])  # and we can always specify stuff if we need to!
```

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

Finally, to capture an arbitrary number of arguments in a function, you can use the `*name` and `**name` parameters. Note that, if both are present, `*name` **must** occure before `**name`, and both must occur after all the formal parameters. When present, the `*name` parameter receives a tuple containing the positional arguments beyond the formal parameter list, and `**name` receives a dictionary containing the key-value pair of the named arguments, except for those corresponding to a formal parameter. For example:

In [9]:
```python
def fun(n, name='Jongbin', *arguments, **keywords):
    """Demo of *name and **name parameters."""
    print '\n' + '=' * 79
    print 'Function called with n=', n,  # note that a leading ',' in
 the print statement will avoid new lines
    print 'Name=', name
    print 'Arguments received:'
    print '\t',  # a tab character to print appropriate indents
    for arg in arguments:
        print arg, '|',
    print '\nNamed arguments received:'
    print '\t',  # a tab character to print appropriate indents
    for key, value in keywords.iteritems():
        print key, '=', value, '|',

fun(1)  # supply minimal arguments
fun(2, 'Padme', 'Amidala', 'Princess', 'testing additional
arguments')  # some additional arguments
fun(2, 'Luke', gender='male', affiliation='Rebel Alliance', text='tes
ting named arguments')  # named arguments
fun(3, 'Anakin', 'Skywalker', 'Jedi', 2015, weapon='Lightsaber', skil
l='force')  # both
```

```
=====================================================================
==========
Function called with n= 1 Name= Jongbin
Arguments received:

Named arguments received:

=====================================================================
==========
Function called with n= 2 Name= Padme
Arguments received:
        Amidala | Princess | testing additional arguments |
Named arguments received:

=====================================================================
==========
Function called with n= 2 Name= Luke
Arguments received:

Named arguments received:
        gender = male | text = testing named arguments | affiliation
 = Rebel Alliance |
=====================================================================
==========
Function called with n= 3 Name= Anakin
Arguments received:
        Skywalker | Jedi | 2015 |
Named arguments received:
        weapon = Lightsaber | skill = force |
```

Sometimes, an opposite situation may occur, where the required arguments are in a list/tuple or keyword arguments are in a dictionary, and you would like to unpack them programatically in the function call. In such cases, you can use the `*name` and `**name` conventions introduced above in the function call. For example:

In [10]:
```
print 'regular call:', range(1, 10, 2)  # the range function takes ar
guments (start, stop[, step])
args = [1, 10, 2]  # pack the arguments (equivalent to above) into a
 list
print 'unpack from list:', range(*args)  # all the function by unpack
ing the list
```

```
regular call: [1, 3, 5, 7, 9]
unpack from list: [1, 3, 5, 7, 9]
```

In [11]:
```
def print_info(name, email, phone):
    """Quick demo of keyword argument unpacking."""
    print 'Name:', name
    print 'email:', email
    print 'phone:', phone

kwargs = {'name':'Jongbin Jung', 'email':'jongbin at stanford.edu',
'phone':'650-123-4567'}
print_info(**kwargs)
```

```
Name: Jongbin Jung
email: jongbin at stanford.edu
phone: 650-123-4567
```

Lambda expressions

Anonymous one-liner functions can be created with the `lambda` keyword wherever function objects are required, but you don't want or need to define a full function.

In [12]:
```
# sort the list of words by the number of vowels in each word,
# but without defining a separate count_vowel function
words = ['queueing', 'management', 'alpha', 'omega']
words.sort(key=lambda word: sum([word.count(v) for v in
'aeiouAEIOU']))
print words
```

```
['alpha', 'omega', 'management', 'queueing']
```

Modules

Once you start building functions, you might want to collect certain functions as a general 'toolbox' to be used across multiple projects. In `python`, you can put definitions in a file with a `.py` extension. Such a file is called a `module`. Once you save your functions into a `module`, you can `import` them. Let's practice with some examples.

For illustration purposes, create let's create two modules that contain one function of the same name each:

In [13]:
```
# save this function to a file named module1.py
def speak():
    """Make module 1 say something"""
    print 'Module 1 speaking ...'
```

In [14]:
```
# save this function to a file named module2.py
def speak():
    """Make module 2 say something"""
    print 'Hi, this is module 2 speaking!'
```

You can import each module (and the functions in them) using the `import` statement as follows:

In [15]:
```
import module1
import module2
```

Note that the name you use in the `import` statement is just the file name of the module, without the `.py` extension.

When you `import` a module, `python` creates an isolated 'space' for each module. This allows different modules to have functions of the same name, without causing confusion. But because of this, when ever you want to use a function from a certain module, you have to specify the module name before calling the function. Compare:

In [16]:
```
module1.speak()
module2.speak()
```

```
Module 1 speaking ...
Hi, this is module 2 speaking!
```

This can be a bit painful (and messy) if your module names get longer. There are typically two ways to work around this:

1. `import` with the `as` keyword to assign your own name to a model
2. assign your own function name to a module's function

Each approach is illustrated below, which to use should depend on the context and personal style:

In [17]:
```
import module1 as m1
import module2 as m2
m1.speak()
m2.speak()
```

```
Module 1 speaking ...
Hi, this is module 2 speaking!
```

In [18]:
```
import module1
import module2

speak1 = module1.speak  # note the lack of parentheses
speak2 = module2.speak  #  when assigning functions to a new name

speak1()
speak2()
```

```
Module 1 speaking ...
Hi, this is module 2 speaking!
```

Finally, modules can also be executed as standalone scripts. However, to do this, the module must know when it's been `imported` or executed. This is done in `python` by specifying a __name__ variable within each module's 'space'. When a module is `imported`, it's __name__ variable is set to the filename it was imported from:

In [19]:
```
module1.__name__
```

Out[19]:
```
'module1'
```

However, if a module is executed, for example from the terminal with the command,

```
python module_name.py
```

then the __name__ variable is set to `__main__`.

To illustrate this, let's create a new module, `module3.py`:

In [20]:
```
# save this code to a file named module3.py
def speak():
    """Make module 3 say something"""
    print 'My __name__ is', __name__


if __name__ == '__main__':
    speak()
    print 'You\'ve executed me!'
```

```
My __name__ is __main__
You've executed me!
```

In [21]:
```
import module3

module3.speak()
```

```
My __name__ is module3
```

Now, instead of `import`ing module3, execute it from a command prompt with the command:

```
python module3.py
```

(you can open a command prompt within `spyder`)

The output should look like:

```
My __name__ is __main__
You've executed me!
```

A little more on `module` execution ...

When executing a module from the command prompt, you can also pass arguments to the module in the form of

```
python filename.py arguments
```

The arguments are passed to the module via a list in the `sys` standard module, and can by accessed by calling `sys.argv`. (*standard* modules are modules that are built-in to `python`). The first (position 0) element of `sys.argv` contains the execution call of the module, so arguments that are passed through the command prompt start from position 1.

For example, we can write a module that takes a single argument from the command prompt as follows:

In [22]:
```
# save this code to a file named module4.py
import sys  # import the standard module sys

if __name__ == '__main__':
    print 'The first element of sys.argv is', sys.argv[0]
    print 'The argument passed was:', sys.argv[1]
```

```
The first element of sys.argv is /usr/lib/python2.7/site-packages/ipy
kernel/__main__.py
The argument passed was: -f
```

Then, execute from the command prompt with an argument, for example:

```
python module4.py hello
```

This should print to the screen:

```
The first element of sys.argv is module4.py
The argument passed was: hello
```

Note that all arguments are passed as a string by default. If you want to use a different type, you will have to convert it within `python`, e.g., `int(sys.argv[1])` to convert the first argument into an integer).

(if you want to do some serious argument parsing, you should take a look at the `argparse` module (https://docs.python.org/2/howto/argparse.html))

... and while we're on the topic of the sys standard module

Quite often, you'll want use `python` to build tools that will be chained within a pipeline, e.g., you'd like to do something like

```
cat some_file.txt | python your_script.py > some_output.out
```

The `sys` module lets you read from the system `stdin`, as if it were a file object, i.e., you can loop through each line of `sys.stdin`. As a simple example, let's write a `python` script that will print lines from `stdin` that contain the word `two` (case sensitive):

In [23]:

```
# save this code to a file named module5.py
import sys

for line in sys.stdin:
    line = line.rstrip('\n')
    if 'two' in line.split():
        print line
```

The, we can extract all the lines in our `two_cities.txt` file with the simple chained command

```
cat two_cities.txt | python module5.py
```

(notice how fast that was!), and (as an example) we can count how many of those lines there are by adding the `wc` shell command:

```
cat two_cities.txt | python module5.py | wc -l
```

Exercise 3.

Continuing from the previous exercise ...

1. Write a function `top_n(d, n=5)`, which takes a dictionary of word counts (such as that created in the previous exercise) and an optional argument `n`, and prints words that have the top `n` count, along with the actual count.
    - to sort a dictionary by its values, use the built-in function `sorted(iterable, cmp=None, key=None, reverse=False)`; you can set the sorting `key` to the dictionary's value by setting `key=d.get`, and sort in descending order by setting `reverse=True`.
    - you might want to create a simple word count dictionary to test your function
2. Expand your function from the previous question into a module that can be executed with a target filename and integer `n` as an argument, i.e.,
   `$ python module_name.py target_file.txt 5`
   which,
    A. reads the contents of the target file
    B. generates a word occurence count dictionary from the text
    C. prints word/count of the words with top `n` occurences
3. (bonus) Can you modify the script to read input from `stdin`, instead of a specified file?