week2_lecture

July 13, 2020

1 Obtaining Course Materials

The course materials will be updated throughout the course, so we recommend that you download the most recent version of the materials before each lecture or recap session. The latest notebooks and other materials for this course can be obtained by following these steps:

1. Go to the github page for the course: https://github.com/semacu/data-science-python
2. Click on the green "Clone or download" button, which is on the right of the screen above the list of folders and files in the repository. This will cause a drop-down menu to appear:
3. Click on the "Download ZIP" option in the drop-down menu, and a zip file containing the course content will be downloaded to your computer
4. Move the zip file to wherever in your file system you want the course materials to be held e.g. your home directory
5. Decompress the zip file to get a folder containing the course materials. Depending on your operating system, you may need to double-click the zip file, or issue a command on the terminal. On Windows 10, you can right click, click "Extract All...", click "Extract", and the folder will be decompressed in the same location as the zip file
6. Launch the Jupyter Notebook app. Depending on your operating system, you may be able to search for "Jupyter" in the system menu and click the icon that appears, or you may need to issue a command on the terminal. On Windows, you can hit the Windows key, search for "Jupyter", and click the icon that appears:
7. After launching, the Jupyter notebook home menu will open in your browser. Navigate to the course materials that you decompressed in step 5, and click on the notebook for this week to launch it.

2 Functions

2.1 Overview

A function is basically a block of code that only runs when it is called. Functions are a general programming tool, and can be found in many programming languages. All functions operate in three steps: 1. Call: the function is called with some inputs 2. Compute: based on the inputs, the function does something 3. Return: the function outputs the result of the computation

Python has many functions pre-loaded, for example the print function. In this example, the function is **called** with some text as the input, the **computation** is doing something to the text (keeping it the same), and the **return** is the printing of text to screen.

[1]:
```
print("I just called the print function")
```

```
I just called the print function
```

As well as the input, some functions also have additional options which change how the function works. These additional options are called **arguments**. For example, the print function has a *sep* argument which allows the user to specify which character to use as the text separator.

[2]:
```
print("By", "default", "text", "is", "separated", "by", "spaces")
print("We", "can", "choose", "to", "separate", "by", "dashes", sep="-")
```

```
By default text is separated by spaces
We-can-choose-to-separate-by-dashes
```

Some useful functions in Python are: - type() : returns the variable type for an object - len() : returns the length of a list or tuple - abs() : returns the absolute values of a numeric variable - sorted() : returns a sorted copy of a list or tuple - map() : applies a function to every element of a list or tuple, and returns a corresponding list of the returns of the function

2.2 Defining your own function

2.2.1 Why?

Functions allow you to wrap up a chunk of code and execute it repeatedly without having to type the same code each time. This has four main advantages: 1. Accuracy: by typing the code once, you only have one chance to make a mistake or typo (and if you discover a mistake, you only have to correct it once!) 2. Flexibility: you can change or expand the behaviour of a function by making a change in the function code, and this change will apply across your script 3. Readability: if someone reads your code, they only have to read the function code to understand how it works, instead of reading chunks of repeated code 4. Portability: you can easily copy a function from one script and paste it in another script, instead of having to extract general-purpose lines of code that are embedded in a script written for a specific purpose

2.2.2 How?

To declare a new function, the **def** statement is used to name the function, specify its inputs, add the computation steps, and specify its output. This results in a **function definition**, and the function can then be called later in the script. All function definitions must have three elements: 1. Name: you can call your function anything, but usually verbs are better than nouns (e.g. echo_text rather than text_echoer). Python will not check whether a function already exists with the name that you choose, so **do not use the same name as an existing function** e.g. print() 2. Input: almost all functions take at least one input, either data to perform computaton on, or an option to change how the computation proceeds. You don't have to specify an input, but you **must** add a set of brackets after the function name, whether this contains any inputs or not 3. Return: the

return statement is required to signify the end of the function definition. You don't have to return anything at the end of the function, but the return statement must still be there

Below is a simple example of a function that takes one input (the text to be printed), and returns the text to the user. We assign the results of calling the function to a new variable, and print the results. As you can see, every line that you want to include in the function definition must be indented, including the **return** statement:

[3]:
```
def print_text(input_text):
    output_text = input_text
    return(output_text)

result = print_text("Hello world")
print(result)
```

```
Hello world
```

2.3 Variable scope

A key difference between code inside a function and elsewhere in a script is the **variable scope**: that is, where in the script variables can be accessed. If a variable is declared outside a function, it is a **global variable**: it can be accessed from within a function, or outside a function. In contrast, if a variable is declared inside a function, it is a **local variable**: it can only be accessed from inside the function, but not outside. The code below demonstrates this - the *counter* variable is declared inside the function, to keep track of the number of words that have been checked, and it is accessed without trouble by the print message inside the function:

[4]:
```
# this function checks whether each word in a list of words contains a Z
def check_Z(word_list):
    Z_status = []
    # this counter is a local variable
    counter = 0
    for i in word_list:
        counter += 1
        if "Z" in i:
            Z_status.append(True)
        else:
            Z_status.append(False)
        # accessing the local variable inside the function works fine
        print("Word {} checked".format(counter))
    return(Z_status)

check_Z(["Zoo", "Zimbabwe", "Ocelot"])
```

```
Word 1 checked
Word 2 checked
Word 3 checked
```

[4]:
```
[True, True, False]
```

However, if we add an extra line at the end of the script which attempts to access the *counter* variable outside the function, we get an error:

[5]:
```
# this function checks whether each word in a list of words contains a Z
def check_Z(word_list):
    Z_status = []
    Z_progress = []
    # this counter is a local variable
    counter = 0
    for i in word_list:
        counter += 1
        Z_progress.append(counter)
        if "Z" in i:
            Z_status.append(True)
        else:
            Z_status.append(False)
        # accessing the local variable inside the function works fine
        print("Word {} checked".format(counter))
    return((Z_status, Z_progress))

check_Z(["Zoo", "Zimbabwe", "Ocelot"])
# accessing the local variable outside the function doesn't work
print("Final counter value = {}".format(counter))
```

```
Word 1 checked
Word 2 checked
Word 3 checked


      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-5-7fed6def616f> in <module>
        16 check_Z(["Zoo", "Zimbabwe", "Ocelot"])
        17 # accessing the local variable outside the function doesn't work
   ---> 18 print("Final counter value = {}".format(counter))


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

As this shows, variables declared inside a function are effectively invisible to all code that is outside the function. This can be frustrating, but overall it is a very good thing - without this behaviour, we would need to check the name of every variable within every function we are using to ensure

that there are no clashes!

2.4 Advanced functions

2.4.1 Setting defaults for inputs

For some functions, it is useful to specify a default value for an input. This makes it optional for the user to set their own value, but allows the function to run successfully if this input is not specified. To do this, we simply add an equals sign and a value after the input name. For example, we may want to add a *verbose* input to the **print_text()** function to print progress messages while a function runs.

[6]:
```
def print_text(input_text, verbose=False):
    if verbose:
        print("Reading input text")
    output_text = input_text
    if verbose:
        print("Preparing to return text")
    return(output_text)
```

The default setting for the *verbose* input is *False*, so the behaviour of the function will not change if the *verbose* input is not specified:

[7]:
```
print_text("Hello world")
```

[7]:
```
'Hello world'
```

But we now have the option to provide a value to the *verbose* input, overriding the default value and getting progress messages:

[8]:
```
print_text("Hello world", verbose=True)
```

```
Reading input text
Preparing to return text
```

[8]:
```
'Hello world'
```

If we specify a default value for any input, we must put this input after the other inputs that do not have defaults. For example, the new version of **print_text()** will not work if we switch the order of the inputs in the function definition:

[9]:
```
def print_text(verbose=False, input_text):
    if verbose:
        print("Reading input text")
    output_text = input_text
    if verbose:
        print("Preparing to return text")
    return(output_text)
```

```
print_text("Hello world", verbose=True)
```

```
  File "<ipython-input-9-8ab60e4151f1>", line 1
    def print_text(verbose=False, input_text):
                   ^
SyntaxError: non-default argument follows default argument
```

2.4.2 Functions calling other functions

One useful application of functions is to split a complex task into a series of simple steps, each carried out by a separate function, and then write a master function that calls all of those separate functions in the correct order. Having a series of simple functions makes it easier to In this way, an entire workflow can be constructed that is easy to adjust, and scalable across a large number of datasets. For example, we might have a number of sample names, and for each we want to: 1. Find the longest part of the name 2. Check whether that part begins with a vowel 3. Convert the lowercase letters to uppercase, and vice versa

Once we have done this, we want to gather the converted sample names into a list.

[10]:
```
def find_name_longest_part(input_name):
    parts = input_name.split("-")
    longest_part = ""
    for i in parts:
        if len(i) > len(longest_part):
            longest_part = i
    return(longest_part)

# print(find_name_longest_part("1-32-ALPHA-C"))

def check_start_vowel(input_word):
    vowels = ["a", "e", "i", "o", "u"]
    word_to_test = input_word.lower()
    if word_to_test[0] in vowels:
        return(True)
    else:
        return(False)

# print(check_start_vowel("ALPHA"))

def convert_case(input_word):
    output_word = ""
    for i in input_word:
        if i.islower():
```

```
            output_word += i.upper()
        else:
            output_word += i.lower()
    return(output_word)

# print(convert_case("ALPha"))

def convert_sample_name_parts(sample_names):
    converted_names = []
    for i in sample_names:
        longest_part = find_name_longest_part(i)
        if check_start_vowel(longest_part):
            converted_names.append(convert_case(longest_part))
    return(converted_names)

convert_sample_name_parts(["1-32-ALPHa-C", "1-33-PHI-omega", "1-34-BETA-sigMA"])
```

[10]: ['alphA', 'OMEGA']

2.5 External functions

2.5.1 Importing modules

One of the great things about Python is the enormous number of functions that can be loaded and used. By default, Python only loads a basic set of functions when it is launched (or when a Jupyter notebook is opened), but extra functions can be loaded at any time by importing extra **modules**. To import a module, use the **import** command. For example, we can import the **os** module, which contains a number of functions that help us to interact with the operating system:

[12]:
```
import os
```

Once a module has been imported, all of the functions contained within that module are available to use. For example, the **os** module has a **getcwd()** function, which returns the current working directory. To call this function, we must specify the module name and the function name together, in the form MODULE.FUNCTION:

[13]:
```
os.getcwd()
```

[13]: 'C:\\Users\\kmlm215\\OneDrive - AZCollaboration\\pydatasci\\data-science-python\\notebooks'

Some of the most useful data science modules in Python are not included with a standard Python installation, but must be installed separately. If you are using conda, you should always install modules through conda as well.

2.5.2 Importing specific functions from modules

Some modules are very large, so we may not want to import the whole module if we only need to use one function. Instead, we can specify one or more functions to be loaded using the **from** command:

[14]:
```
from os import getcwd
```

If a function has been loaded using this syntax, it does **not** need to be prefaced with the module name:

[15]:
```
getcwd()
```

[15]:
```
'C:\\Users\\kmlm215\\OneDrive - AZCollaboration\\pydatasci\\data-science-
python\\notebooks'
```

2.5.3 Aliasing

Sometimes it is useful to be able to refer to a module by another name in our script, for example to reduce the amount of typing by shortening the name. To do this, we can use the **as** command to assign another name to a module as we import it, a technique known as **aliasing**. For example, we can import the **numpy** module and alias it as **np**:

[16]:
```
import numpy as np
```

Now whenever we want to call a function from this module, we use the alias rather than the original name. For example, the **zeros()** function allows us to create an array of specific proportions, with each element as a zero:

[17]:
```
np.zeros((3,4))
```

[17]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

We can also alias a function name if we import it using the **from** command. However, as with naming your own function, it is critical to avoid using the name of a function that already exists:

[18]:
```
from os import getcwd as get_current_wd
get_current_wd()
```

[18]:
```
'C:\\Users\\kmlm215\\OneDrive - AZCollaboration\\pydatasci\\data-science-
python\\notebooks'
```

2.5.4 What not to do

In some code, you may see all of the functions being imported from a module using the following syntax:

[19]:
```
from os import *
```

At first glance this looks more convenient than importing the module as a whole, because we no longer need to preface the function name with the module name when calling it:

[20]:
```
getcwd()
```

[20]:
```
'C:\\Users\\kmlm215\\OneDrive - AZCollaboration\\pydatasci\\data-science-
python\\notebooks'
```

However, **YOU SHOULD NEVER DO THIS**. This is because if the module contains a function with the same name as a function that is already loaded, Python will replace that pre-loaded function with the new function **without telling you**. This has three consequences: 1. The pre-loaded function is no longer available 2. Debugging is now much more difficult, because you don't know which function is being called 3. If you call the function and expect to get the pre-loaded version, you will get unexpected (and potentially disastrous) behaviour

3 Exercises

3.1 Exercise 1

In the code block below, there is a list of 5 protein sequences, specified in the single amino acid code where one letter corresponds to one amino acid. Write a function that finds the most abundant amino acid in a given protein sequence, but prints a warning message if the protein sequence is shorter than 10 amino acids. Run your function on each of the proteins in the list.

[ ]:
```
proteins = [
    "MEAGPSGAAAGAYLPPLQQ",
    "VFQAPRRPGIGTVGKPIKLLANYFEVDIPK",
    "IDVYHY",
    "EVDIKPDKCPRRVNREVV",
    "EYMVQHFKPQIFGDRKPVYDGKKNIYTVTALPIGNER"
]
```

3.2 Exercise 2

The code below is intended to specify a function which looks up the capital city of a given country, and call this function on a list of two countries. However, it currently has a bug which stops it running. There are three possibilities for the nature of this bug: 1. Its arguments are in the wrong order 2. It uses a variable that is out of scope 3. It is missing the return statement

What is the bug?

[ ]:
```
capital_cities = {
    "Sweden": "Stockholm",
    "UK": "London",
```

```
    "USA": "Washington DC"
}

def find_capital_city(verbose=True, country):
    if country in capital_cities:
        capital_city = capital_cities[country]
        if verbose:
            print("Capital city located")
    else:
        capital_city = "CAPITAL CITY NOT FOUND"
    return(capital_city)

countries = ["USA", "UK", "Sweden", "Belgium"]
for i in countries:
    print(find_capital_city(i))
```

3.3 Exercise 3

In the data folder, you will find a file ”imagine_lyrics.txt”, which contains the lyrics to the song Imagine by John Lennon. Your task is to find out which word is used most frequently in the lyrics. There are many ways to approach this, but however you solve it, remember to **break up your code into functions!**

[ ]:

3.4 Exercise 4

Some words aren't very interesting (”the”, ”a”, ”and” etc), so we might want to exclude these from consideration when finding the most frequent word in a set of lyrics. Extend your code from Exercise 3 to include an option to exclude a custom list of words, and test how the results change when excluding ”the”, ”a” & ”and”.

[ ]: