Python Basics

This is part 1 of the Python portion of a Software Carpentry workshop that is built around some light handling of Twitter data. There are four parts. At the end of this section participants will have produced a chart that shows word counts for a given set of text. Skills covered include:

- Jupyter cells
- Comments
- Variables
- Functions and methods
- Coops
- Conditionals
- Control Structures
- Operators
- Matplotlib
- User input

In this first section we'll quickly cover some basic syntax, control structures, objects, and output in Python so that when it comes to looking at the basics of actual analysis we are in a better position to see the big picture.

Boxes and Markdown

Open `Python-Lesson1.ipynb`

Let's begin by typing a brief description of what we are about to do into a *markdown* box at the top of a new Python 3 workbook in Jupyter notebooks. Enter something like the following in a markdown box.

```
Learning the basics of Python.  When done I will have created my own word histogram.
```

Note that while markdown is raw text on input an interpreter will convert various character sequences into formatting commands that will change the final output. Change the original line to say:

```
**Learning the basics of Python.**  When done I will have created *my own* word count
er.
```

It will continue to look just as you typed it until you either `hold shift and press enter` or press the `play button` to process the block. Do either now and your text will render as:

Learning the basics of Python. When done I will have created *my own* word counter.

Note that you can double click on the block to edit the text and then process the block again.

We won't do any more here but you are encouraged to play and experiment on your own later.

There in no single standard to appeal to for markdown formatting although there is general agreement on some core behaviours. When working within Jupyter you should refer to the Markdown section of the Jupyter docs. You may also find it useful to refer to the GitHub Markdown Guide since that is another popular place where Markdown is used. If you are looking for a template document this guide was written in markdown and should be available wherever you got the PDF version as a .md file.

Note that there are other types of cells/blocks as well. Make sure that the right cell type is turned on for the content or you won't get the results you are expecting.

Comments

In the cell immediately below the markdown description we just made is where we'll build our histogram tool. Make sure that the type of the cell is code.

Before we write any actual code we should plan out our program. We'll do this with comments. Comments are portions of the code that are not interpreted by the language when the program is run. Typically comments are helpful descriptions of surrounding code for humans to read. Sometimes they are used during testing to turn on and off lines used in debugging.

There are two types of comments in Python, single line and multi line. Single line comments start with a #, the hash or pound symbol found above the numeral 3 on most keyboards. Let's enter the following single line comments to roughly describe what we are going to be doing:

```
# This program will count the words in a string and graph the 10 most common words

# Take some input

# Process the input word by word, counting each word

# Produce the chart
```

Multi line comments are openend and closed with triple double quotes: """. We don't have a use for them now but we likely will before the workshop is over.

Let's create or move to a new cell below this one where we can learn and practice how to use the pieces of the python language that we will need to create our word histogram.

Functions

Type the following into the new code block.

```
print("hello world!")
```

Notice that the text is coloured. This is the Python interpreter running in the background telling you what various components of the statement are.

See what the line does by running the cell.

```
hello world!
```

Congratulations, you have just written and run your first program!

Let's look at what happened here. "print" is what is known as a function, meaning that it is a sort of list of commands or processes that the computer is to perform. Functions are always invoked by writing out the name of the function followed by a set of parentheses. What is inside the parenthesis are the inputs into the function, what it needs or expects to carry out the instructions inside. There can be many inputs, just one (as is the case here), or no inputs (such functions are known as "empty functions"). Here our input is the text string `hello word!`. We know that it is a string of text characters because we have wrapped it in quotations.

> What happens when the quotations are:
>
> a. removed?
>
> b. swapped with single quotes?
>
> c. used inconsistently (e.g. no closing quote, open with a single but closed with a double, etc.)?
>
> d. doubles are used inside singles (e.g. " 'hello wold!' ") or vice versa?
>
> e. the quote type used inside is the same as is used outside (e.g. " "hello wold!" ")

[Go over errors and show how to escape characters with `\`. If they have done the commandline part of this already then they should be reminded that they have seen this before in reference to spaces in filenames.]

Note that print() is quite happy to take more than one input:

```
print("hello","world","!")
```

or no input

```
print()
```

IMPORTANT: A major change from Python 2 to Python 3 is the shift from print just being a statement:

```
print "hello world!"
```

to print being implemented as a function (for consistency):

```
print("hello world!")
```

It is a small change that may not seem to matter but it really does because it renders the vast majority of Python 2 programs incompatible with Python 3.

print() is a built-in function, it comes pre-installed and activated with every version of Python. There are other functions that are built in but which we must load, others that we will have to install separately and then load, and still others that we'll create ourselves.

For the moment though we'll leave these further details of functions behind and look at another core concept, variables.

Variables

Variables provide us named containers to hold our objects making it easier to pass the contents around within the program without actually having to provide the original object. For a parallel in language consider how convenient it is to use the phrase "my parents" to refer to people who are my parents rather than carrying them around with me everywhere and having to point to them each time I refer to them. Let's not get carried away with the analogy.

In the case our hello world example imagine how annoying and counterproductive it would be if what we were working with was not "hello world!" but the text of Moby Dick!

Let's create and use some variables. We'll start by typing the following in a new block:

```
firstVar = "hello world!"
```

What we have done here is create a variable called "myVar" and assign to it the string "hello world!". Three important points to note:

1. **The variable name is short and meaningful within this context.** It is the first variable that we have created in python. Avoid the temptation to call your variables in other programs things like "var1" or "x"

because doing so will make it much harder to debug the program since the variable name does not even hint at what it is holding.
2. **The equal sign is the assignment operator and *not* the equality operator.** When invoked it assigns the name on the left to the object on the right.
3. **Running this code block will result in no output.** As with the command line "silence is golden" so we'll only see output when we explicitly ask for it (there are a few exceptions in Jupyter, such as when we have only a variable name on a line and its content gets printed out) or deserve it. Hence the importance of print().

> How do we view the content of our variable?

We can assign many different types of data to a variable, even changing the type after the variable is created (this is not true with other "strict typing" languages). Let's assign it a numerical value:

```
firstVar = 5
```

We are going to need a variable that holds a string to run our word counting tool so let's do create that now in our master block that we will use throughout.

```
# Take some input
inputString = "this is some sample text"
```

Control Structures Part 1: Loops

Now that we have some content to use as a draft we need to process it. To do this we'll need to make use of both loops and conditionals. We'll start with loops.

Loops are used when we want to do the same thing over and over again, typically with changes to at least one of the inputs. This allows us to code more efficiently. In Python there are two types of loops, *while loops* and *for loops*. While loops run until a condition is met. For loops run until an entire list is consumed. The exception to both these descriptions is that both loops can be ended using a `break` statement within the loop.

Let's look at an example of each:

- While

  fruit = 'banana' index = 0 while index < len(fruit): letter = fruit[index] print(letter) index = index + 1

- For fruit = 'banana' for letter in fruit: print(letter)

[Walk through each of these line by line with the attendees.]

As you can see, for loops are usually the more compact and efficient choice when the entire list is to be processed. Do not forget the while loop though, it is the correct choice when you don't know where to stop processing a list.

> Note that the only thing special about the variable names used is that they are meaningful for humans. Prove this to yourself by changing each of 'fruit', 'index', and 'banana' to something totally different within each code block. Make sure to do this *consistently*.

If we want to count *all* the words is a given body of text then we should use a for loop. We should also test to make sure that we are reading the input properly so let's print the result of each pass through the loop:

```
# Process the input word by word, counting each word
for word in inputString:
    print(word)
```

> What happens when we run this code? Why?

The problem is that Python is seeing our input as a single string of characters rather than a list of words (Note though that this is further proof that the variable names only matter to us). We could fix this now but let's finish the control logic of the program first, thinking of this as *character* histogram for the moment.

Lists and containers, including strings

Lists

As we move through the loop we are going to need to collect the items we come across---whether they are characters or words or anything else---and count them. To do this we need to use containers and lists. Let's have a quick primer on lists and containers.

- Ordered collection of objects. strings are a special type of list where each element is a character and the list type has its own methods.

  list1 = ["hello","world","!"]

- indexing and slicing

> If `[x]` is the indexing command then what will `list1[1]` return? Why?
>
> What about `list1[1:2]`?
>
> What will `list1[0]="Hello"` do?

- Lists can hold almost any type of object and can be heterogeneous.

  `list2=[5,4.0,"wow"]`

Strings

Strings are *ordered sequences of characters* and as sequences we can index and slice strings just like lists. We saw this happening with the loop examples and this is what is happening with our little program right now.

```
string1 = "Hello World!"
```

> Why do we get "e" as the return? Zero based counting.

```
 string1[0:5]
```

> What values will just give us "world!"? `string1[6:12]` and `string1[6:]`
>
> What about `[0:0]` and `[6:3]` and negative values?

Strings are immutable. We modify them only by overwriting them.

Dictionaries

The container type that we will use for our little program though is a dictionary. Dictionaries use key-value pairs for indexing rather than locations in a list.

```
 dict1 = {'one':'un','two':'deux','three':'trois'}
```

> Display the contents of the dictionary

- note how the order has (likely) changed!
- Accessing elements

  ```
  dict1[two]
  ```

- *dict1[two]* fails, no variable named "two"

  ```
  dict1['two']

  dict1['four']
  ```

- *dict1[‘four’]* fails, no index named "four"
- Checking members

```
‘four’ in dict1

‘trois’ in dict1
```

- Adding/modifying members

```
outFile = open(‘output.txt’, ‘r’)

for line in outFile:

  print linedict1[four] =  ‘quatre’
```

- Checking for inverse members. Can’t do that yet (Dictionaries don’t work that way by default). For that we need functions, loops, and conditionals.

We will use a dictionary that uses the word/letter as the key and returns a count of that letter. For the moment let's just make sure that we can load the dictionary and worry about actually counting later:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
histdict = {}
for word in inputString:
    histdict[word] = 1

print(histdict)

> Note that both histdict and the print statement are outside the loop.  What will ha
ppen if we move either or both of them inside the loop?  Try it.
```

What we need now is a way to count the occurrences of a word/letter. To do this we need to be able to recognize when we have seen an item before or not and then change the behaviour accordingly. To do this we need our second control structure: conditionals.

Control Structures Part 2: Conditionals

The core conditional control structure in Python is the if. It is pretty straightforward to use, as follows:

```
x = 5

y = 4

if x < y:

    print ("x is less than y")
```

Of course this is not the only case, y can also be less than x. We can add this as follows:

```
x = 5
y = 4

if x < y:
    print ("x is less than y")
else:
    print ("y is less than x")
```

Still, we have not caught all the cases

```
x = 5
y = 4

if x < y:
    print ("x is less than y")
elif x > y:
    print ("y is less than x")
elif x == y:
    print("x is equal to y")
else:
    print("Something weird is happening")
```

While it seems unnecessary to add the last line because all the cases are covered it is good practice to explicitly trap all the cases you expect and then use else to grab any that you don't.

We can also combine conditionals with logical operators

```
x = 5
if 0 < x and x < 10:
    print("x is a positive single digit number")
```

We can now go back and modify our program by adding a conditional that will allow us to initialize an entry for an item to 1 when we see it for the first time or to do something else if we've seen it before:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString:
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] = 2

print(dicthist)
```

Of course, we need some way to account for items that occur more than twice. For this we need to talk about operators.

Operators

Objects are the nouns of the language. You do things to them with other things. Strings, integers, variables, lists, etc. are all types of objects. Operators are the things you use to do things to objects. When you do something to an object with an operator we refer to that object as an operand. Things like + - ^ * ** / = == < > <= >= are all operators.

You've already seen punctuation, these are the characters (including spaces) that keep the syntax together. These include spaces, : , () [] {}

Expressions arise when we link objects together with operators and the right punctuation.

Statements are things that the computer will do. Expressions are kinds of statements but think of them as separate things. Expressions have values, statements work with values.

> Create some variables and assign numbers to them. Try all the standard operators + * / - on them.
>
> What does 1 // 2 produce? Why? Integer division.

IMPORTANT: In Python 2 a `/` is *integer division by default*. Can overcome this by adding decimal placeholders to at least one component of the formula.

> What does the `**` operator do?

- ** is the exponent. Often ^ elsewhere.

```
2 ** 3
```

> What does the % operator do?

- % is the modulo operator. It returns the remainder when the first number is divided by the second.

```
7 % 2
```

- returns 1 since 2 goes into 7 three times with one left over.

> What operators can we use with lists? Try some out!

```
  list1 + list2

  list1 * 3

  list1 + string1
```

Note that *list1 + string1* fails. Need to recast string1 as a member of a list:

```
list1 + [string1]
```

With operators we can modify our program to continually add each new instance of an item that it discovers:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString:
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] = dicthist[word] + 1

print(dicthist)
```

Alternatively, we can use the increment operator to make this cleaner:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString:
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] += 1

print(dicthist)
```

It works! For letters anyway. Let's see how to get it working for words.

Methods

As we have seen, the program as currently written treats our input as a string, which is a sort of list that has an ordered set of characters as its members. What we need to do is turn it into a list. We could write a program that would read through the string adding each character to a new string until finding the end ofa word, adding that word to a list, and then continuing the process until everything is done and *then* processing this new list rather than the initial string. Ugh. That's a lot of work. Fortunately someone else has already done this for us. Even more fortunately they have done it for lots of similar actions and there is an easy way to invoke them, methods.

Methods are special functions that are associated with certain data types. They are invoked by typing the name of the variable holding that data type and then immediately following that with a dot and then the name of the method.

Let's try some methods out on some lists and some strings.

- list methods. append, expand, and sort. Note that list methods are all void. They modify the original list and return None.

  list1.sort()

  list1.reverse

  list1.index("Hello")

  list[1].capitalize()

list1[1]=list[1].capitalize()

- Adding

  list1.append(list2)

[Once append is on the table it is possible to demonstrate the difference between passing by reference (append) and passing by value (+). Note as well that appending a list to a list will make the appended list a *member of* the first list while using + will make all its members members.

```
list1 = ["hello","world","!"]
list2 = [5,4.0,"wow"]
list3 = list1 + list2
list4 = list1
list4.append(list2)
print(list3)
print(list4)
list1[0]="HELLO"
print(list3)
print(list4)
```

]

Need to know the available methods? Tab/Shift-Tab completion.

- How will we capitalize the *"wow"* in list1?

  ```
  list1[3][2].capitalize()

  list1.insert(2,"two")
  ```

- deleting elements

  ```
  list1.pop(2)

  list1.remove("wow")
  ```

- *list1.remove("wow")*fails. Need to access the list in the list.
- Only removes the first instance
- string methods: len, upper, lower, replace, etc.

```
len(string1)

string1.swapcase()

string1.swapcase()
```

- Why didn't it go back to lowercase the second time? We're not modifying the string.

```
string1.replace("!", "!!!")
```

- *'trois' in dict1* fails because it is a value, not a key

```
dict1Val = dict1.values()

'trois' in dict1Val
```

- Or go straight for the throat with *'trois'* in *dict1.values()*

What we want for our program is the split method:

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString.split():
    if word not in dicthist:
        dicthist[word] = 1
    else:
        dicthist[word] += 1

    print(dicthist)
```

> How do we prevent the dictionary from printing every time?

- Remove the indent in front of `print` so that it is no longer associated with the for loop.

> Why might we want to print the dictionary every time?

- Testing, it allows us to see what is built and how.

> Is there input that can "break" our tool? If that is too vague then try: Find some input that seems to violate our ability to count the same word multiple times. Specifically, where what we would sensibly considered

> the same word is counted as multiple words by the tool.

- Capitalization of duplicated words causes trouble. We'll solve thi

Now that we have our data in a useful format we can look at plotting it as a histogram

Charts

The core library used for plotting in Python is matplotlib. It has website that is full of examples, some of which might be exactly what you want: matplotlib.org. Ultimately it is a very powerful package that can quickly become quite complex to use (We could easily spend a day just on it). Extra help is available from the External Resources Page. We'll glance at the basics, drop in a solution, and then move on (You'll be seeing it again).

Let's start by playing with the following example:

```
import matplotlib.pyplot as plt

#special ipython/jupyter command that keeps the output in this window rather than ope
ning another one.
%matplotlib inline

plt.figure()
plt.plot([1, 2, 3, 4], [10, 20, 25, 30], color='lightblue', linewidth=3)
plt.scatter([0.3, 3.8, 1.2, 2.5], [11, 25, 9, 26], color='darkgreen', marker='^')
plt.bar([1,2,3,4],[12,3,25,18], width=0.2, align='center')
plt.xlim(0.5, 4.5)
plt.ylim(0,50)
plt.show()
```

> See what happens as you change the values around, comment lines out, and change the options.

Note three important things: 1. The very first line of this block *imports* a new library. The library is matplotlib. Actually, the whole library isn't loaded, just a portion of it called "pyplot". This portion is also renamed "plt". 2. A special ipython/jupyter only command is issued that stops the plot from appearing in its own window. The percentage sign at the beginning indicates that it is this type of command.

> How could you print out the line, bar, and scatter plot separately but from the same cell?

It is time for us to add a bar chart to our program. We can copy up a lot of content from the example. We'll add one feature that will allow us to include labels in the plot, making the whole program look like this.

```
# Take some input
inputString = "this is some sample text"

# Process the input word by word, counting each word
dicthist={}
for word in inputString.split():
if word not in dicthist:
    dicthist[word] = 1
else:
    dicthist[word] += 1

print(dicthist)

import matplotlib.pyplot as plt
%matplotlib inline

plt.figure()
plt.bar(range(len(dicthist.values())),dicthist.values(), width=0.2, align='center')
plt.xticks(range(len(dicthist)), list(dicthist.keys()),rotation='vertical')

plt.show()
```

User Input

At times it can be really useful to take user input. We can do this with the input() function, changing the inputString line to:

```
inputString = input("Please enter some sample text")
```

Tweaking

Figure getting a little small? Consider changing its dimensions by swapping `plt.figure(figsize=(15, 6))` for `plt.figure()`.

Too much information on the chart? Set it to only print the top 10 most common items:

And that should be enough of the basics. You've seen all the primary pieces of python at this point *and* make a working program.