

ArcGIS automation through scripting with Python

3_Basics

- Hello World! 2
- Importing packages 4
- Variables and data types 5
- Iteration 7
- Using Functions to block code 9

Step 1: Hello World!

Our first foray into Python, and for many programmers, is the simple “Hello World!” script, we will make a play on this by using this for many examples of the different basic programming tasks that we will undertake.

You have already used basic print functions, but we will revisit these again, as print is one of the most useful built in functions that we will use. It can tell us a lot about the current processing status and you can use it to provide helpful messages for yourself or other users.

But before we get to print, you will benefit greatly from ensuring that you take the time to produce well **commented** code.

Download the *Github > Classes > 3_Basics > Step_1.py* document

Comments in Python are indicated by a # (hash) symbol, any characters on a line that begins with a hash will be treated as a comment. In PyCharm, this will appear as grey text:

```
# The hash symbol is used to create a single line comment.  
  
# You can use  
# multiple lines  
# if you want
```

What and how you comment, entirely depends on your needs (sometimes, if you are using contributed code, it is useful to cite the source as a comment).

PyCharm has some useful features, such as block commenting:

```
You can also turn  
a block of code  
into a comment by  
using PyCharm:  
1. Select your block of code.  
2. Select "Code" from the menu bar.  
3. Select "Comment with Line Comment".  
4. You can also reverse this using the same steps 1-3.
```

Try it and see, you can also use the Keyboard shortcut: CTRL + Forward Slash.

Some of you may have come across block comments:

```
# Block comments in the form of something like  
# /*
```

```
#      This is a JavaScript style block comment
# */
```

Comments of that style are not used in Python, use a hash symbol for each line that you wish to comment.

Example:

Please comment the following code that I have provided, and then execute the code:

```
# Commenting style is very user dependent, I want you to attempt to comment
# the small code block below. What kind of information do you feel is pertinent?

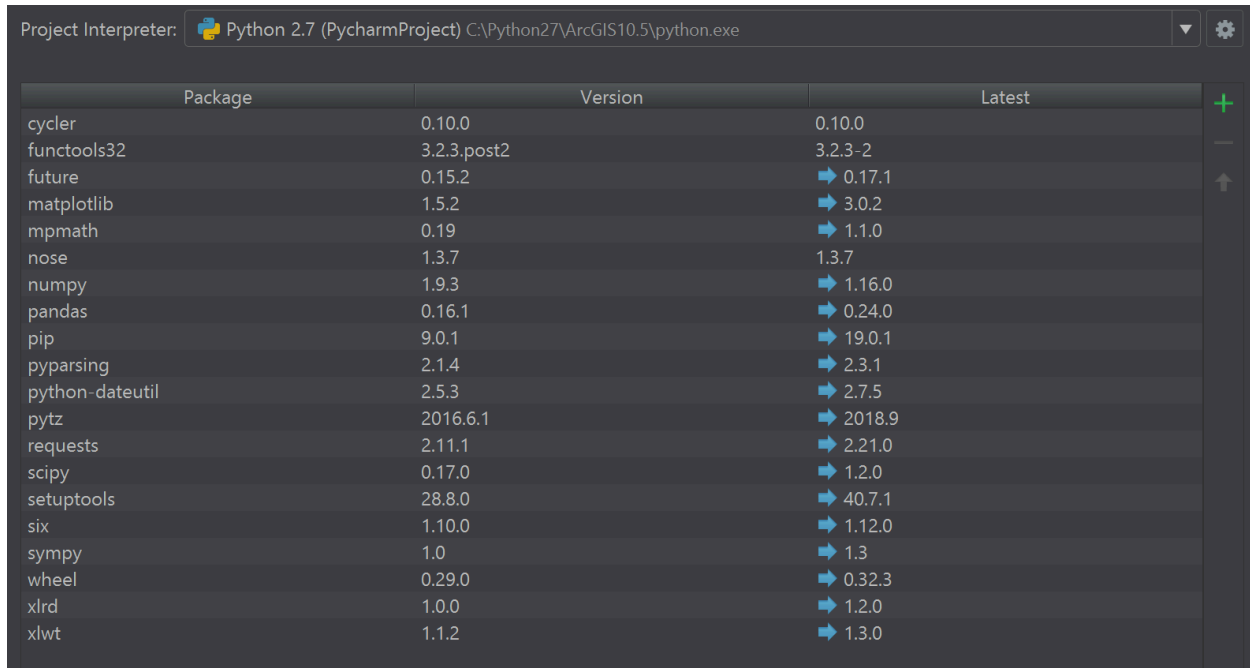
list = ['Hi', 'Hello', 'Allo']
list2 = ['Bye', 'Goodbye', 'Au Revoir']

for i in list:
    for z in list2:
        print i + ' but its not ' + z
```

To reiterate, good commenting is essential for you maintaining your sanity, whilst also producing usable code.

Step 2: Importing packages

One powerful ability of Python is in the ability to add *packages* to the installation which can give you additional functionality incredibly quickly. Arcpy/ArcGIS comes with some built in packages as default (note that this method will not show all packages):



The screenshot shows the 'Project Interpreter' window in PyCharm. The title bar indicates 'Python 2.7 (PycharmProject) C:\Python27\ArcGIS10.5\python.exe'. The window contains a table with three columns: 'Package', 'Version', and 'Latest'. The table lists various installed packages and their latest available versions. On the right side of the table, there are icons for adding (+), removing (-), and refreshing (↻) the package list.

Package	Version	Latest
cycler	0.10.0	0.10.0
functools32	3.2.3.post2	3.2.3-2
future	0.15.2	➡ 0.17.1
matplotlib	1.5.2	➡ 3.0.2
mpmath	0.19	➡ 1.1.0
nose	1.3.7	1.3.7
numpy	1.9.3	➡ 1.16.0
pandas	0.16.1	➡ 0.24.0
pip	9.0.1	➡ 19.0.1
pyarsing	2.1.4	➡ 2.3.1
python-dateutil	2.5.3	➡ 2.7.5
pytz	2016.6.1	➡ 2018.9
requests	2.11.1	➡ 2.21.0
scipy	0.17.0	➡ 1.2.0
setuptools	28.8.0	➡ 40.7.1
six	1.10.0	➡ 1.12.0
sympy	1.0	➡ 1.3
wheel	0.29.0	➡ 0.32.3
xlrd	1.0.0	➡ 1.2.0
xlwt	1.1.2	➡ 1.3.0

You can see this by going to *File > Settings > Project ###Name### > Project Interpreter*. You can also query this from the Python command line or in a Python script:

```
help("modules")
```

Of course, there are many more that are available, and you can add these into your Python27/ArcGIS 10.5 install if you need or want to. I do caution, that if you aim to share your code, you will have to provide instructions for users on how to install your additional required packages as many will not have a clue, particularly those that are using Python Toolboxes.

To import packages, there are several ways of doing so:

```
# Importing packages is easy:
import arcpy
arcpy.AddMessage(r'arcpy loaded!\n')

# You can also change the name of the package you load into something else:
import arcpy as arcpython
arcpython.AddMessage(r'I changed the name of arcpy\n')

# You can import more than one package at once:
import arcpy, os
```

```
arcpy.AddMessage(os.listdir(r'c:'))

# You can also be selective in your loading of various packages by using from:
from arcpy import AddMessage
arcpy.AddMessage(r'I only loaded AddMessage')
```

Step 3: Variables and data types

Variables are how you store information, they store items in memory that you will use later in your script. There are various naming conventions for variables, and you should choose the best practice for yourself.

For example:

```
# This is bad variable naming:  
x = r'C:\Shapefile.shp'
```

These are examples of invalid variable naming:

```
# This is not valid, as variables cannot start with a number nor can they contain  
spaces:  
1x = r'C:\Shapefile.shp'  
x 1 = r'C:\Shapefile.shp'
```

I tend to use either “camelCase” or underscored words, you may also wish to include the variable type in the name, for example:

```
# Best practice variable naming:  
input_file_path = r'C:\Shapefile.shp'  
inputFilePath = r'C:\Shapefile.shp'  
inputFilePathString = r'C:\Shapefile.shp'
```

The more information you use, the more you are likely to be able to track your variable through your code. Of course, it can get unwieldy, which is why PyCharm is useful as it can autocomplete variable names using “Tab” button.

Moving on to data types, python can handle several main data types:

STRING

Strings are a primary datatype in Python and can hold any kind of character:

```
# String  
example_string = 'this is a string, it can store anything... Well almost'  
example_string = 'you cannot use the same 'quote' mark within a string'  
example_string = 'but you can use this: "quote"'  
example_string = "or this 'quote'"  
  
example_string = 'escaping slashes is also an issue, \''  
example_string = r'using the r letter (raw string) prior to the string ignores it-> \''
```

```
# You can declare a string using str()
example_string = str('this is a string, but I would be anyway without str()')

# You can join strings -
example_join_string = example_string + ' ' + '.. Magic!'
# But you can only join strings, this will fail:
example_join_string = example_string + ' ' + 2 + '.. Magic!'
example_join_string = example_string + ' ' + str(2) + '.. Magic!'
```

INTEGER

Integers are whole numbers, without any decimals, cannot contain letters, you should be wary when using int() for calculation:

```
# Integer
example_integer = 2
# You can also declare an integer type-
example_integer = int(2)
not_an_integer = int(str('I am not an int'))
# Be wary when computing using int:
5 / 2 # = 2.5? Not according to Python, as it uses the floor value, and doesn't change
datatype to float
```

FLOAT

Floats are floating point values, and are essentially decimal. Floats should always be used when performing a calculation, but often you can fall into a trap where you are not, leading to an accumulation of error when integers are used..

```
# Floats
5.0 / 2 #As the first value is float, it will work, and give the correct answer
5 / 2.0 #As the second value is float, it will work, and give the correct answer
# You can use round to round numbers, the first argument is the number, and the second
the decimal places:
round(2.858, 2)
# You can convert int to float to allow it to be used in calculation
float(int(2))
```

LISTS

Lists are an ordered datatype useful for storing information, as you can easily iterate through a list. Any data type can be embedded within a list, including other lists.

```
# Lists

# You can initiate a blank list:
my_blank_list = []

# Or populate your list with variables
my_first_list = ['item 1', 'item 2', 'item 3', 4, 5., ['a', 'b']]
```

```

# You can easily iterate through your list, see what happens to your list within a
list though
for i in my_first_list: print i
# You can query an item within a list:
print my_first_list[0] #Note that zero notation, 0 = first item in list

# You can add items to an exisiting list
my_first_list.append('You added me later!')

# You can also merge two lists:
list_1 = ['a', 'b', 'c']
list_2 = ['alpha', 'beta', 'gamma']
list_1.extend(list_2)

# You can also order lists
number_list = [1, 3, 4, 2]
number_list.sort() #Ascending
print(number_list)
number_list.sort(reverse=True) #Descending
print(number_list)

# You may remove items from a list, by index, slice or by variable content
del number_list[0]
print number_list

del number_list[1:2]
print number_list

number_list.remove(1)
print number_list

```

TUPLES

Tuples are a datatype that is immutable, it cannot be changed after creation, use normal brackets to create (). Otherwise, they behave much like lists (indexing, iteration etc):

```

# Tuples
my_first_tuple = (1, 2, 3)
print my_first_tuple

# Tuples are immutable, cannot be changed after creation
my_first_tuple.append(4)

```

DICTIONARIES

Dictionaries are defined by using curly brackets ({}) and are used to create key:value pairs. They are useful as you can refer to a key, which may have several additional pairs. Think of a shapefile OID, plus other values from other columns:

```

# Dictionaries
my_first_dictionary = {'Key1':'Hi', 'Key2':'Bye'}
print my_first_dictionary['Key1']

# You can't access a value or a key using an index:

```



```
print my_first_dictionary[0]

# More complex dictionary:
my_second_dictionary = { 1 : [{'Distance': '23', 'Density':'9'}], 2 : [{'Distance':
'50', 'Density':'10'}], 3 : [{'Distance': '12', 'Density':'3'}] }
print my_second_dictionary[1]
print my_second_dictionary[1][0]['Distance']
print my_second_dictionary[1][0]['Density']
```

Key value pairs are used in data storage types such as JSON and are a common output of many API's and Databases.

Step 4: Iteration

You can iterate many different datatypes in Python, let's start with a string, and go right through, using a *for loop*:

```
# For Loops

# For loop on string
my_string = 'abcde'
for i in my_string:
    print i

# For loop on list
my_list = ['a', 'b', 'c', 'd', 'e']
for i in my_list:
    print i

# For loop on tuple
my_tuple = ('a', 'b', 'c', 'd', 'e')
for i in my_tuple:
    print i

# For loop on dictionary
my_dictionary = {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
for i, j in my_dictionary.items():
    print i, j
```

If/else/elif

```
# You can use if/else/elif to make decisions based on your data:
my_var = 4
if my_var > 6:
    print 'Greater than 6'
elif my_var == 5:
    print 'Equal to 5'
else:
    print 'Less than 5'

# Using if to catch cases as we loop:
my_if_list = [1, 2, 3, 4, 5, 6]
for i in my_if_list:
    if i % 2 == 0:
        print str(i) + ' is even because we can divide it by two'
    else:
        print str(i) + ' is odd because it will not divide by two'
```

While

```
# We can use while to perform some task
i = 0
while i < 10:
```

```
    i = i+1
    print i

# Be careful though not to enter an infinite loop as you forgot your increment
i = 0
while i < 10:
    print i
```

Step 5: Code cleanliness

Python is indentation sensitive, which means you need to pay attention to your code to avoid indentation errors:

```
IndentationError: expected an indented block
```

```
# This will trigger an indentation error, but luckily your error will be helpful in
figuring out where the fix is
i = 0
while i < 10:
i = i+1
print i
```

You can use spaces or tabs, tabs are preferred as you don't need to remember how many you are using for indentation.

Fix the following code:

```
# Fix the following code, using the appropriate indentation, you want to produce the
following result:
# n = 1
# n = 2
# n = 3
# n = 4
# n = 5 <---
# n = 5
# n = 6 <---
# n = 6
# n = 7

try:
n = 1
while n < 10:
if n == 5:
print 'n = 5 <---'
else:
if n == 6:
print 'n = 6 <---'
print 'n = ' + str(n)
n = n + 1
except:
print 'my code failed'
```

Step 6: Using Functions to block code

Functions are a great way of repeatably packaging code:

```
# A simple function:
def square(x):
    y = x ** 2
    return y

print square(3)
print square(8347329)
```

They can also take multiple arguments:

```
# Function with multiple arguments
def string_parser(string, n):
    while string:
        yield string[:n]
        string = string[n:]
print list(string_parser("1234567890",2))
```