Introduction to Python

Python basics and installation

Python content and code examples credits

The slides with python content and code snippets are taken from

- https://learnpython.org
- https://www.tutorialspoint.com/python/index.htm

Overview

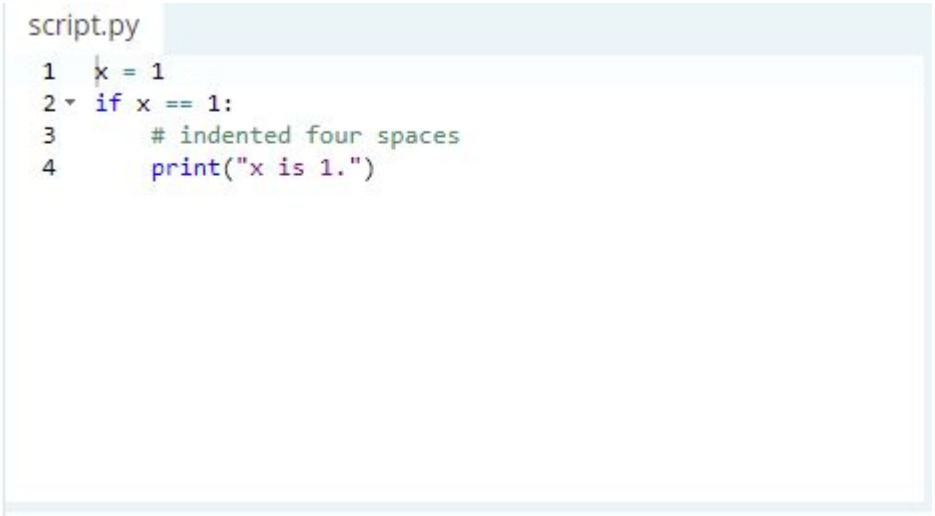
Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

It provides very high-level dynamic data types and supports dynamic type checking.

Indentation

Python uses indentation for blocks, instead of curly braces. Both tabs and spaces are supported, but the standard indentation requires standard Python code to use four spaces. For example:



Variables and Types

Python is completely object oriented, and not "statically typed". You do not need to declare variables before using them, or declare their type. Every variable in Python is an object. The types of variables are

Numbers

Python supports two types of numbers - integers and floating point numbers. (It also supports complex numbers, which will not be explained in this tutorial).

script.py

```
1  myint = 7
2  print(myint)
```

IPython Shell

```
In [1]:
```

Run

Powered by DataCamp

To define a floating point number, you may use one of the following notations:

script.py

```
1  myfloat = 7.0
2  print(myfloat)
3  myfloat = float(7)
4  print(myfloat)
```

IPython Shell

```
In [1]:
```

Variables and Types

Strings

Strings are defined either with a single quote or a double quotes. The difference between the two is that using double quotes makes it easy to include apostrophes (whereas these would terminate the string if using single quotes)

script.py

```
1  mystring = "Don't worry about apostrophes"
2  print(mystring)
```

Assignments can be done on more than one variable "simultaneously" on the same line like this

script.py

```
1  a, b = 3, 4
2  print(a,b)
```

Lists

Lists are very similar to arrays. They can contain any type of variable, and they can contain as many variables as you wish. Lists can also be iterated over in a very simple manner. Here is an example of how to build a list.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];

list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

You can update and delete list elements, list = ['physics', 'chemistry', 1997, 2000];
Update the list element 1, list[1] = 2020
Delete the list element 2, del list[2]

Refer to https://www.tutorialspoint.com/python/python_lists.htm to learn more about lists

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

Accessing values in tuples is similar to accessing values in lists, `tup1[0]: physics` and `tup2[1:5]: [2, 3, 4, 5]`

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples

```
#!/usr/bin/python
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
# Following action is not valid for tuples
# tup1[0] = 100;
# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3;
```

Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example `dict['Age'] = 8; # update existing entry`

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Functions in python are defined using the block keyword "def", followed with the function's name as the block's name. They may also receive arguments from a caller and return a value to caller.

script.py

```
1   # Define our 3 functions
2   def my_function():
3       print("Hello From My Function!")
4
5   def my_function_with_args(username, greeting):
6       print("Hello, %s , From My Function!, I wish you %s"%
    (username, greeting))
7
8   def sum_two_numbers(a, b):
9       return a + b
10
11  # print(a simple greeting)
12  my_function()
13
14  #prints - "Hello, John Doe, From My Function!, I wish you
    a great year!"
15  my_function_with_args("John Doe", "a great year!")
16
17  # after this line x will hold the value 3!
18  x = sum_two_numbers(1,2)
```

Basic Operators

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Refer to https://www.tutorialspoint.com/python/python_basic_operators.htm

Decision Making

script.py

```
statement = False
another_statement = True
if statement is true:
    # do something
    pass
elif another_statement is true: # else if
    # do something else
    pass
else:
    # do another thing
    pass
```

Loops

There are two types of loops in Python, for and while.

For loops iterate over a given sequence. Here is an example:

script.py

```
1  primes = [2, 3, 5, 7]
2  for prime in primes:
3      print(prime)
```

For loops can iterate over a sequence of numbers using the "range" function

```
1    # Prints out the numbers 0,1,2,3,4
2    for x in range(5):
3        print(x)
```

While loops repeat as long as a certain boolean condition is met. For example:

```
1  # Prints out 0,1,2,3,4
2
3  count = 0
4  while count < 5:
5      print(count)
6      count += 1  # This is the same as count = count + 1
```

break and continue statements

break is used to exit a for loop or a while loop, whereas **continue** is used to skip the current block, and return to the "for" or "while" statement. A few examples:

```
# Prints out 0,1,2,3,4

count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break

# Prints out only odd numbers - 1,3,5,7,9
for x in range(10):
    # Check if x is even
    if x % 2 == 0:
        continue
    print(x)
```

Classes and Objects

Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.
A very basic class would look something like this:

```
class MyClass:
    variable = "blah"

    def function(self):
        print("This is a message inside the class.")

myobjectx = MyClass()
myobjecty = MyClass()

myobjecty.variable = "yackity"

# Then print out both values
print(myobjectx.variable)
print(myobjecty.variable)
# accesing the object function
myobjectx.function()
```

List Comprehensions

List Comprehensions is a very powerful tool, which creates a new list based on another list, in a single, readable line.
For example, let's say we need to create a list of integers which specify the length of each word in a certain sentence, but only if the word is not the word "the".

```
1  sentence = "the quick brown fox jumps over the lazy dog"
2  words = sentence.split()
3  word_lengths = []
4  for word in words:
5      if word != "the":
6          word_lengths.append(len(word))
7  print(words)
8  print(word_lengths)
```

Using a list comprehension, we could simplify this process to this notation:

```
1  sentence = "the quick brown fox jumps over the lazy dog"
2  words = sentence.split()
3  word_lengths = [len(word) for word in words if word !=
   "the"]
4  print(words)
5  print(word_lengths)
```

Modules and Packages

In programming, a module is a piece of software that has a specific functionality. For example, when building a ping pong game, one module would be responsible for the game logic, and another module would be responsible for drawing the game on the screen. Each module is a different file, which can be edited separately.

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):
   print "Hello : ", par
   return
```

The import Statement: You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax - `import module1[, module2[,... moduleN]`.

Modules and Packages

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

The from...import Statement: Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax − `from modname import name1[, name2[, ... nameN]]`

For example, to import the function fibonacci from the module fib, use the following statement − `from fib import fibonacci`

Exception Handling

When programming, errors happen. It's just a fact of life. Perhaps the user gave bad input. Maybe a network resource was unavailable. Maybe the program ran out of memory. Or the programmer may have even made a mistake! Python's solution to errors are exceptions.

But sometimes you don't want exceptions to completely stop the program. You might want to do something special when an exception is raised. This is done in a *try/except* block.Here's a trivial example: Suppose you're iterating over a list. You need to iterate over 20 numbers, but the list is made from user input, and might not have 20 numbers in it. After you reach the end of the list, you just want the rest of the numbers to be interpreted as a 0. Here's how you could do that:

```
1  def do_stuff_with_number(n):
2      print(n)
3
4  def catch_this():
5      the_list = (1, 2, 3, 4, 5)
6
7      for i in range(20):
8          try:
9              do_stuff_with_number(the_list[i])
10         except IndexError: # Raised when accessing a non
   -existing index of a list
11             do_stuff_with_number(0)
12
13 catch_this()
```

There, that wasn't too hard! You can do that with any exception. For more details on handling exceptions, look no further than the [Python Docs]

Install Miniconda and Spyder

Miniconda can be installed for Python 2 (32 bit or 64 bit) or Python 3 (32 bit or 64 bit) depending on which ArcGIS environment it will be integrated with

Download miniconda from https://docs.conda.io/en/latest/miniconda.html

Install Miniconda and open the miniconda prompt

To create a user local env: `conda create –n <env-name> python=2.7.10`
OR
To create a multi-user local env: `conda create –n <env-path> python=2.7.10`

Example: conda create -n arc1041 python=2.7.10 numpy=1.9.2 matplotlib=1.4.3 scipy=0.16.0 pandas pyparsing xlrd xlwt console_shortcut spyder

Install Pycharm

Download and install pycharm https://www.jetbrains.com/pycharm/

References

https://www.learnpython.org/

https://www.tutorialspoint.com/python/index.htm