Python

2020 DSS Bootcamp

Shawn Santo

08-03-20

Supplementary materials

Companion videos

1. Python basics and data structures
2. Control flow
3. Functions
4. Classes, methods, and attributes
5. Libraries

Preliminaries

Before you get started

In order to actively follow along you have a few options.

- Use a docker container from Duke OIT

    1. Go to https://vm-manage.oit.duke.edu/containers
    2. Log in with your Duke NetID and password
    3. Find `Jupyter - interactive data science and scientific computing notebooks`
    4. Click the link to create your environment

- Get Python and Jupyter notebook for your own computer

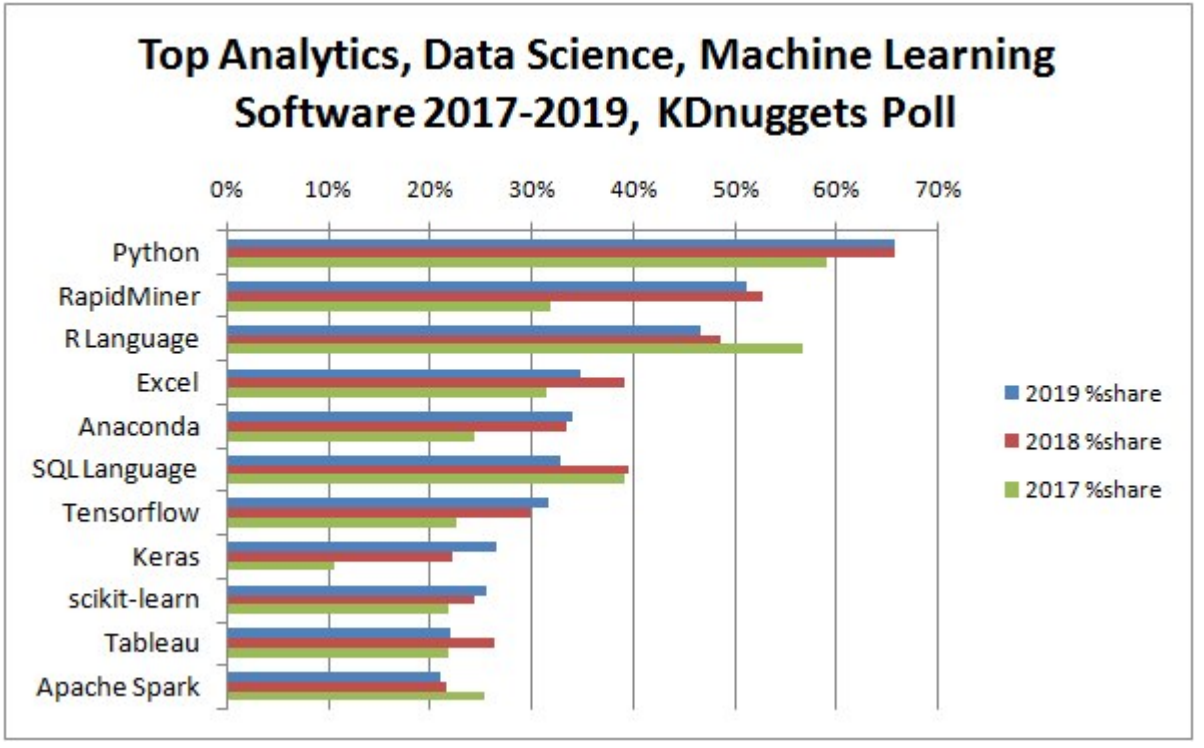
    1. Go to https://www.anaconda.com/distribution/
    2. Download Python 3.x version based on your OS
    3. Follow install instructions at https://docs.anaconda.com/anaconda/install/

- Use RStudio with R Markdown (run Python, but no Jupyter notebook)

    1. Log in to the DSS RStudio server
    2. In an R Markdown file you can insert Python chunks instead of R chunks
    3. Click the dropdown arrow next to insert

Why Python?

KDnuggets poll



Source: https://www.kdnuggets.com/2020/06/data-science-tools-popularity-animated.html

Stack Overflow trends

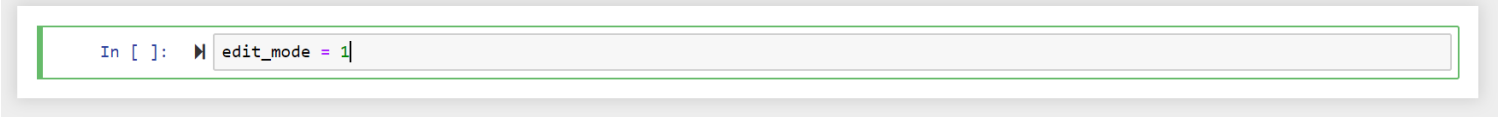
To see how technologies have trended over time based on use of their tags since 2008 we can look at Stack Overflow trends.

Jupyter notebook

Overview of the notebook

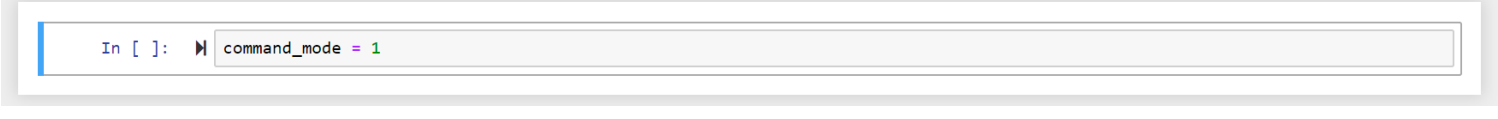
Bimodal interface: edit mode and command mode

Click in a cell or hit `enter` to enter edit mode



When in edit mode you can type code or write text with markdown.

Hit `esc` to enter command mode



When in command mode you can make edits to the notebook, but not individual cells.

Notebook shortcuts

In **edit** mode:

- Run cell and add new cell: `shift + enter`
- Add a line within a cell: `enter`

In **command** mode:

- Save the notebook: `s`
- Change cell to markdown: `m`
- Change cell to code: `y`
- Cut, copy, paste, delete a cell: `x`, `c`, `v`, `d`
- Add a cell above, below: `a`, `b`

The point-and-click interface is also an option to execute these commands.

Jupyter notebook versus R Markdown

- Similar to R Markdown, Jupyter notebooks allow you to write code and text in one easy to read document that is reproducible and easy to share with others.
- A Jupyter notebook does not knit to an HTML, PDF or Word file. However, you can embed HTML into a notebook.
- For a more detailed comparison see The First Notebook War.

Python basics

Arithmetic operators

Python supports the standard arithmetic operations.

```
10 + 8
```

```
18
```

```
3 - 2
```

```
1
```

```
100 / 6
```

```
16.666666666666668
```

```
80 * 0
```

```
0
```

```
100 // 3 # integer division
```

```
33
```

```
13 % 4   # modulo operator
```

```
1
```

```
4 ** 3   # raise to power
```

```
64
```

No need to worry about the difference between `int` *and* `float`*.*

Comparison operators

```
5 > 0
```

```
True
```

```
4 < 100
```

```
True
```

```
3 == 3 # equality
```

```
True
```

```
10 <= (5 * 1.9)
```

```
False
```

```
0 != 1 # not equal
```

```
True
```

Use boolean key words `and`, `or`, and `not` for multiple comparisons.

```
(5 > 10) or (8 / 2 == 4)
```

```
True
```

```
True and (8 > 64 ** 0.5)
```

```
False
```

```
not True and (8 > 64 ** 0.5)
```

```
False
```

```
not (True and (8 > 64 ** 0.5))
```

```
True
```

Parentheses matter. Also, python is case sensitive. True != TRUE unless you set a variable TRUE = True which is something you probably shouldn't do.

Operators `&`, `|`, and `^` are bitwise operators; you probably will not need them. However, these operators are used in R, so try not to get confused.

Python versus R

Key differences in operators in Python versus R.

Operator purpose	Python	R
true	`True`	`TRUE`, `T`
false	`False`	`FALSE`, `F`
or	`or`	`|`, `||`
and	`and`	`&`, `&&`
not	`not`	`!`
modulo	`%`	`%%`
exponentiation	`**`	`^`
integer division	`//`	`%/%`

Object assignment

Use a single = for object assignment.

```
a = 10
a
```

```
10
```

```
b = [-10, 0, "computing", "bootcamp", 5 / 2]
b
```

```
[-10, 0, 'computing', 'bootcamp', 2.5]
```

```
my_string = "This is a longgggg string"
my_string
```

```
'This is a longgggg string'
```

```
nums = list(range(7))
nums
```

```
[0, 1, 2, 3, 4, 5, 6]
```

Shortcut operation and assignment

Unlike R, Python supports things like

```
a = 5
a += 3
a
```

```
8
```

```
val = 10
val **= 3
val
```

```
1000
```

These are equivalent to `a = a + 3` and `val = val ** 3`, respectively.

Data types and structures

Built-in data structures

Python has four primary built-in data structures.

- **List**: an ordered collection of mutable items
- **Tuple**: an immutable list
- **Dictionary**: an unordered collection of key-value pairs
- **Set**: an unordered collection of unique objects

These data structures will generally consist of **integers**, **floating point** numbers, **strings**, or **booleans**. Using the function `type()` will identify the data type or structure.

Hit `p` while viewing the HTML slides for an exhaustive table of the built-in python data types and structures.

Lists

- Lists are ordered sequences that can contain a variety of objects, even other lists.
- A pair of `[ ]` can be used to create a list with each component separated by a comma.
- The first item in a list is considered to be in position 0, not position 1.

```
my_list = [9, [1, 1, 1], 10, "swimming", "hiking"]
my_list
```

```
[9, [1, 1, 1], 10, 'swimming', 'hiking']
```

Indexing

```
my_list[0]
```

```
9
```

```
my_list[-1]
```

```
'hiking'
```

Slicing `start:stop:step`

```
my_list[1:3]
```

```
[[1, 1, 1], 10]
```

```
my_list[1:4:2]
```

```
[[1, 1, 1], 'swimming']
```

Tuples

- Tuples are similar to lists except that they are immutable. You won't be able to reassign something once it exists in a tuple.
- A pair of `( )` can be used to create a tuple. Each component is separated by a comma.

```
cars = ("subaru", "bmw", "hond", "ford")
cars[3]
```

```
'ford'
```

If we try to fix the typo in `cars`, we get

```
cars[2] = "honda"
```

```
TypeError: 'tuple' object does not support item assignment
```

Dictionaries

- Dictionaries are unordered collection of key-value pairs. Each key is connected to a value. The value can be any object you can create in Python.
- Syntax: `{"key_1": value_1, "key_2": value_2}`

```
my_bike = {"model": "trek", "style": "gravel"}
```

Subset based on a key

```
my_bike["model"]
```

```
'trek'
```

Add a dictionary entry

```
my_bike["year"] = 2020
my_bike
```

```
{'model': 'trek', 'style': 'gravel', 'year': 2020}
```

Sets

- Sets are an unordered collection of unique objects that can be created with a pair of `{ }`, where each entry is separated by a comma.
- Test for membership and apply set operators that you know from set theory.

```
my_set = {'a', 'b', 'c', 'd'}
my_set
```

```
{'a', 'c', 'd', 'b'}
```

```
list("seewhathappens")
```

```
['s', 'e', 'e', 'w', 'h', 'a', 't', 'h', 'a', 'p', 'p', 'e', 'n', 's']
```

```
my_set2 = set(list("aabbbbbbbbcddddddde"))
my_set2
```

```
{'c', 'd', 'e', 'b', 'a'}
```

Classes, methods, attributes

A brief introduction to OOP

Python is an object oriented programming (OOP) language. Objects can be categorized by their class type. For each class, there exists a collection of methods and attributes.

- Methods can be thought of as functions defined for a specific class.
- Attributes can be thought of as variables defined for use with respect to a class.
- To access methods and attributes, type `my_object.<method or attribute>`, where after the `.` is the selected method or attribute.

Methods in action

```
my_courses = ["STA523", "STA199", "STA521", "STA601"]
type(my_courses)
```

```
<class 'list'>
```

```
my_courses.append("STA323") # add on an element
my_courses
```

```
['STA523', 'STA199', 'STA521', 'STA601', 'STA323']
```

```
my_courses.reverse() # change the order
my_courses
```

```
['STA323', 'STA601', 'STA521', 'STA199', 'STA523']
```

```
my_courses.pop() # remove last element
```

```
'STA523'
```

```
my_courses
```

```
['STA323', 'STA601', 'STA521', 'STA199']
```

Attributes in action

```
x = 4 + 3j
y = 0 + 1J

[x.real, y. real]
```

```
[4.0, 0.0]
```

```
[x.imag, y.imag]
```

```
[3.0, 1.0]
```

Methods are called similar to functions with `()`; attributes are called similar to a variable or object. Typing `.` and hitting tab will populate the available methods and attributes for a given object.

More methods in action

```
my_string = "python is an object oriented programming language"
```

```
my_string.title() # title case
```

```
'Python Is An Object Oriented Programming Language'
```

```
my_string.count("o") # count the "o"s
```

```
4
```

```
z = (6, 0, 11, 99, -5, 0, 0)
```

```
z.index(11) # find position of 11
```

```
2
```

```
z.count(0)  # count the 0s
```

```
3
```

Exercises

1. Create a float variable and each of the four built-in data structures discussed (list, tuple, dictionary, set). For each object, use a combination of two methods or attributes and document in markdown their functionality.

2. Set object `dilemma`, given below, to all lowercase. Then, reverse the order of the letters. Lastly, capitalize the first "b".

```
dilemma = "Borrow OR Rob"
```

Hint: `dilemma[start:end:step]`. What does a negative index do?

Control flow

`if, for, while`

- The syntax style of control flow in Python is based off `:` and indentations. Python does not wrap code blocks for control flow with `{ }` as is done in R.
- There is no `switch` statement and no `repeat` statement in Python.

if

Syntax:

```
if condition:
  # code block
  # indented
elif another_condition:
  # code block
  # indented
else:
  # code block
  # indented
```

Example:

```
medical_cost = 180

if medical_cost <= 100:
  bill = medical_cost
elif medical_cost <= 200:
  bill = 100 + (0.5 * (medical_cost - 100))
else:
  bill = 150

print("Your total bill is {} dollars.".format(bill))
```

```
Your total bill is 140.0 dollars.
```

`for` loop

Syntax:

```
for index in sequence:
  # code to be
  # iterated
  # again, indented
```

Example:

```
j = 1
for i in "Duke":
  print("Letter {index} is {letter}.\n".format(index = j, letter = i))
  j += 1
```

```
Letter 1 is D.

Letter 2 is u.

Letter 3 is k.

Letter 4 is e.
```

Some more `for` loop examples...

```
squares = []
for i in range(1, 11):
  squares.append(i ** 2)

squares
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
d = {"language": "python", "version": 3.7,
     "year": 1989, "creator": "Guido van Rossum"}

for k, v in d.items():
  print("The key is {}, and its value is {}.".format(k, v))
```

```
The key is language, and its value is python.
The key is version, and its value is 3.7.
The key is year, and its value is 1989.
The key is creator, and its value is Guido van Rossum.
```

`while` loop

Syntax:

```
while condition:
  # code to be
  # iterated
  # again, indented
```

Example:

```
j = 100
i = 1.01
while i < j:
  i *= i
  print(round(i, 2))
```

```
1.02
1.04
1.08
1.17
1.37
1.89
3.57
12.77
163.13
```

Some more `while` loop examples...

```
nums = list(range(3))
while nums:
  num_off = nums.pop()
  print("The number {} was removed from the list nums.".format(num_off))
```

```
The number 2 was removed from the list nums.
The number 1 was removed from the list nums.
The number 0 was removed from the list nums.
```

Python supports `break` statements.

```
my_string = "I'm sleepy, "
i = 1
while True:
  my_string += "z"
  i += 1
  if i > 10:
    break

my_string
```

```
"I'm sleepy, zzzzzzzzzz"
```

Functions

Defining a function

As with control flow, Python relies on indentations and `:` for function style syntax as opposed to `{ }` in R.

```
def func_name(arg1, arg2):
    """
    Docstring starts with a
    short description.

    May have more
    information here.

    arg1 = something
    arg2 = something

    Returns something

    Example usage:

    func_name(1, 2)

    """

    result = arg1 + arg2

    return result
```

```
help(func_name)
```

```
Help on function func_name in module __main__:

func_name(arg1, arg2)
    Docstring starts with a
    short description.

    May have more
    information here.

    arg1 = something
    arg2 = something

    Returns something

    Example usage:

    func_name(1, 2)
```

Functions in action

```
def is_prime(n):
  """
  This function checks if a number is prime.

  n = positive integer

  Example:

  is_prime(2)
  is_prime(16)

  """

  if n < 3:
    result = n - 1
  else:
    for i in range(2, n):
      if (n % i) == 0:
        result = False
        break
    else:
      result = True

  return bool(result)
```

```
help(is_prime)
```

```
Help on function is_prime in module __main__:

is_prime(n)
    This function checks if a number is prime.

    n = positive integer

    Example:

    is_prime(2)
    is_prime(16)
```

```
is_prime(n = 2)
```

```
True
```

```
is_prime(n = 12)
```

```
False
```

```
is_prime(n = 1)
```

```
False
```

```
is_prime(n = 13)
```

```
True
```

Function scope

```
a = 0
b = 1

def my_function():
    print("The value of b here is {}.\n".format(b))
    a = 3
    print("The value of a here is {}.".format(a))

my_function()

print("The value of a here is {}.".format(a))
```

```
The value of b here is 1.

The value of a here is 3.

The value of a here is 0.
```

Exercises

1. Write a function that takes a list and returns a list of the unique elements of the inputted list in descending order. Do this without using `set()`.

```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

```
[5, 4, 3, 2, 1]
```

2. Write a function that counts the number of prime numbers less than or equal to a specified number.

```
prime_count(k = 13)
```

```
6
```

```
prime_count(k = 1000)
```

```
168
```

Mutable versus immutable objects

Object types

- Immutable objects: int, float, long, complex, string tuple, bool
- Mutable objects: list, dict, set, byte array, user-defined classes

Copy made:

```
x = (3, 5, 7)
y = x

id(x) == id(y)
```

```
True
```

```
y = (1, 2, 3)
print(id(x), id(y))
```

```
4755296736 4755338736
```

```
print(x, y)
```

```
(3, 5, 7) (1, 2, 3)
```

No copy made:

```
a = [1, 2, 3, 4]
b = a

id(a) == id(b)
```

```
True
```

```
b.pop()
```

```
4
```

```
print(id(a), id(b))
```

```
4735853264 4735853264
```

```
print(a, b)
```

Python libraries

Libraries of interest

- NumPy: package for scientific computing, handling matrices, arrays, high level mathematical functions
- SciPy: optimization, numerical integration, linear algebra methods
- pandas: data wrangling and analysis (think R's `dplyr`)
- matplotlib: 2D plotting library, generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots
- seaborn: higher level plotting environment, based on matplotlib
- scikit-learn: go to for machine learning in Python, built on NumPy, SciPy, and matplotlib

Importing

A module is a single importable Python file whereas a package is made up of two or more modules. Both can be imported the same way with the keyword `import`.

For example, `import math` imports the math module and provides access to its many functions.

```
import math
math.factorial(5)
```

```
120
```

```
math.sqrt(100)
```

```
10.0
```

```
math.log(1)
```

```
0.0
```

Importing `from`

If you don't want to import all the functions from a module, you can just import the ones you need with the keyword `from`.

```
from math import factorial, sqrt, log
factorial(5)
```

```
120
```

```
sqrt(100)
```

```
10.0
```

```
log(1)
```

```
0.0
```

This technique eliminates the need to preface each function call with `math.`.

Importing `from`

You can import all functions from a given module by using the `*` operator. For example,

```
from math import *
```

imports all the functions from the `math` module.

However, this is a bad idea as you may have conflicts between your object names and function names in the module.

Importing `as`

For brevity, people often give an alias to a package or module on import. This can be done with the keyword `as`. For example,

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
```

Thus, rather than type `matplotlib.pyplot.hist()` a user can type `plt.hist()`. There are common conventions for aliases so stick to using those.

Example

Import packages / modules:

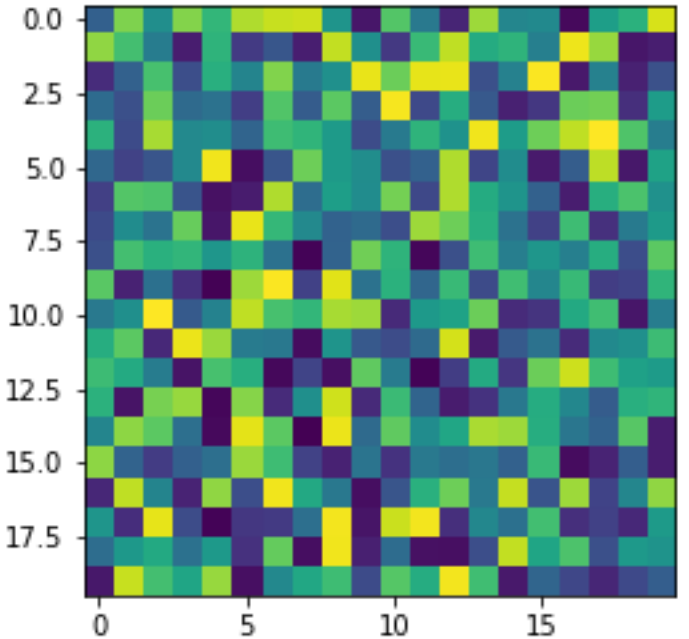
```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
```

In Jupyter notebook you'll want to include the below line of code in a cell to allow for inline plotting.

```
%matplotlib inline
```

Let's plot a heatmap based on a 20 by 20 grid of random numbers.

```
data = np.random.rand(20, 20)
heat = plt.imshow(data)
plt.show()
```



Example

Create a histogram based off of 5000 randomly generated standard normal variables.

```
norm_rv = st.norm.rvs(loc = 0,scale = 1, size = 5000)
n, bins, patches = plt.hist(x = norm_rv, bins = 20, alpha = .5, color = '
          edgecolor = "black", linewidth = 1.2)
plt.show()
```

Exercises

1. Create a histogram based off of random variables generated from a probability distribution of your choice.

2. Plot 100 time steps of a simple symmetric random walk that starts at the value 0.

Related resources

Python programming resources

- Style
  - PEP 8: standard Python style
  - PEP 257: documentation conventions
- Beginner
  - Python: official documentation and tutorial
  - Jupyter: web notebook interface, reproducible research
  - A Byte of Python
  - Python Crash Course
  - Python Crash Course - Cheat Sheets
- Advanced
  - Python Package Index
  - Problem Solving with Algorithms and Data Structures using Python
- Miscellaneous
  - Python 3 Module of the Week

References

1. Introduction · GitBook. (2020). Retrieved from https://python.swaroopch.com/

2. Project Jupyter. (2020). Retrieved from https://jupyter.org/

3. Python Crash Course by ehmatthes. (2020). Retrieved from https://ehmatthes.github.io/pcc/

4. Welcome to Python.org. (2020). Retrieved from https://www.python.org/