

STAT 177, CLASS 01

Richard Waterman

July 2020

OBJECTIVES

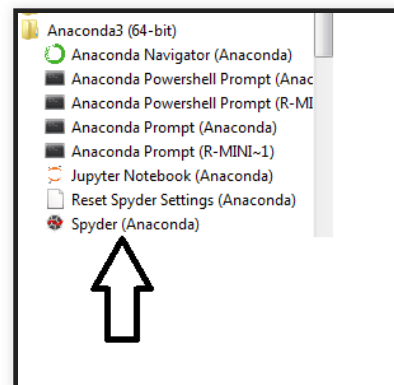
OBJECTIVES

- Opening the Spyder IDE:
 - Introduction to interactive Python.
 - Getting help in Python.
- Types of data:
 - The programming view.
 - The statistical view.
- Storing data in variables.
- Data structures.

USING THE SPYDER IDE

SPYDER

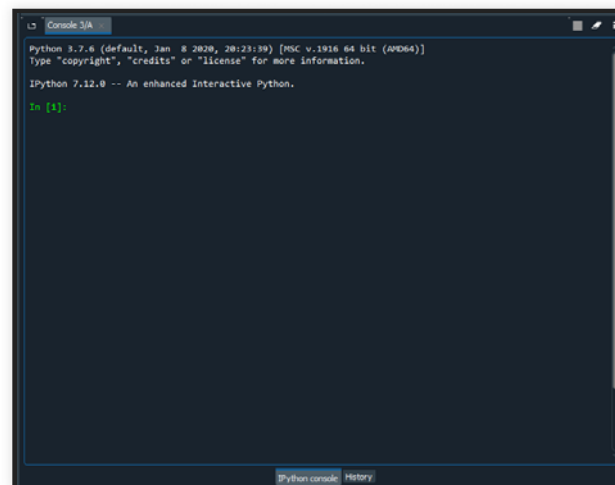
- The Anaconda Python distribution comes with the Spyder IDE (Integrated Development Environment).
- The IDE can make your coding experience more productive and less painful!
- Here's where to find Spyder on a Windows computer:



Spyder location

THE INTERACTIVE PYTHON CONSOLE

- At the bottom right hand side of the IDE is the *interactive Python console* .
- You can “play around” with Python in the terminal.
- This is what the console looks like before typing in any commands:



The image shows a screenshot of a terminal window titled "Console 3/A". The text inside the terminal reads: "Python 3.7.6 (default, Jan 8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)]", "Type 'copyright', 'credits' or 'license' for more information.", "IPython 7.12.0 -- An enhanced Interactive Python.", and a green prompt "In [1]:". At the bottom of the window, there are tabs labeled "Python console" and "History".


USING THE CONSOLE AS A CALCULATOR

- We can do basic math operations in the console, using it like a calculator.
- Try typing the following commands into the console, each followed by the *enter* key.

```
# Addition
2 + 2
# Subtraction
8 - 12
# Multiplication
4 * 3
# Regular division
15 / 6
# Integer division (How many times 6 goes into 15)
15 // 6
# The remainder term from division
15 % 6
# Raising a number to the power
2 ** 10
# Finding the square root of a number
```

9.0

WHAT YOU SHOULD GET AS OUTPUT



```
Python 3.7.6 (default, Jan 8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.12.0 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: 2 + 2
Out[1]: 4

In [2]: 8 - 12
Out[2]: -4

In [3]: 4 * 3
Out[3]: 12

In [4]: 15 / 6
Out[4]: 2.5

In [5]: 15 // 6
Out[5]: 2

In [6]: 15 % 6
Out[6]: 3

In [7]: 2 ** 10
Out[7]: 1024

In [8]: 81 ** 0.5
Out[8]: 9.0

In [9]:
```

First output

MORE COMPLICATED MATH FUNCTIONS

- The most common mathematical functions used in data analysis are the *natural log* and the *exponential* functions.
- We'll just type these into the console and hope for the best.

```
log(10)
```

```
-----  
NameError                                Traceback (most recent call last)  
  
<ipython-input-2-0a816b14e1a2> in <module>  
----> 1 log(10)  
  
NameError: name 'log' is not defined
```

WHY THE ERROR?

- As you can see the `log` function is not found, which is because it is not a part of the Python core.
- But it (along with many other math functions) are available from the built-in Python *math* module.
- You can get additional functionality in Python by using *modules* .
- A *module* is like another Python file that contains a set of functions and definitions and can be read into (imported) the current IPython session.
- The syntax to include a module is

```
import module_name
```

IMPORTING THE MATH MODULE

```
#import the math module  
import math  
# But this still does not work.  
log(10)
```

```
-----  
NameError                                Traceback (most recent call last)  
  
<ipython-input-3-bfbbba626ea05> in <module>  
      2 import math  
      3 # But this still does not work.  
----> 4 log(10)  
  
NameError: name 'log' is not defined
```

```
# But this one does! You need to tell Python in which module to find the function  
math.log(10)
```

```
2.302585092994046
```

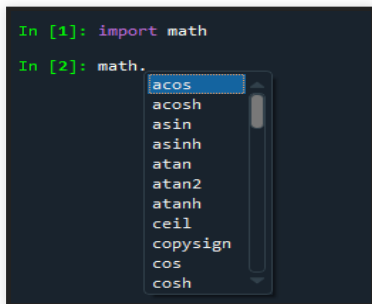
SOME OTHER USEFUL MATH FUNCTIONS

- `math.log` # The natural log
- `math.log10` # The log base 10
- `math.exp` # The exponential function
- `math.ceil` # Round up to the next integer
- `math.floor` # Round down

TAB COMPLETION AND HELP

TAB COMPLETION

- Once you have imported the math module you can see all the functions inside it by using *tab completion* .
- In the console type “math.” and then press the tab key. Here’s what you will see:



```
In [1]: import math
In [2]: math.
acos
acosh
asin
asinh
atan
atan2
atanh
ceil
copysign
cos
cosh
```

- You don’t have to remember all the available functions.
- And you can also use tab completion more generally in IPython to search for available variables and functions.
- Try typing in just the letter *a* , followed by a tab. You will see everything on the search path that starts with *a* .

GETTING HELP ABOUT VARIABLES

- Typing in the question mark “?” before or after a variable gives you some help on it.
- Typing in two question marks “??” may give even more information.
- And syntax like

`math.*y*?`

would list all functions in the math package that contained the letter “y”.

```
math.log10?
```

```
Signature: math.log10(x, /)
Docstring: Return the base 10 logarithm of x.
Type:      builtin_function_or_method
```

DATA TYPES

DATA TYPES

- Data isn't always numeric, and Python has other "types" to handle this.
- Numeric data can be of an integer (whole number) or float (not a whole number) type.
- Text data is called "string" type.
- Logical variables take on the values of True or False and are called "boolean" type.
- There are other special data types for dates and times that we will see later.
- You can convert between variable types, which is called "casting".
- Using the function "type()" will tell you the type of the object you are looking at.
- Using the function "isinstance" will check to see if your variable is of a specific type.
- There is also a special type called "None", which is known as a null value and you can think of as indicating that there is nothing there.

INVESTIGATING WHICH TYPE A VARIABLE IS

- The key command here is ‘type()’

```
# An integer type
type(6)
# A float type
type(3.14)
# A string type -- you can use single or double quotes to delineate a string
type("Hello Stat Students!")
# A boolean type
type(True)
```

```
bool
```

CHANGING A VARIABLE'S TYPE

- The type name can be used as a function to change (cast) a variable's type.

```
# Cast between variable types  
int(13.123) # Turn this number into an integer  
str(98.4) # Turn this number into a string  
float(3) # Turn this integer into a float  
int('4') # Turn the string, 4, into an integer  
int(True) # Turn a logical into an integer
```

1

CHECKING IF A VARIABLE IS OF A PARTICULAR TYPE

- The key command is 'isinstance'

```
# Check out the type of a variable
a = 6
print(isinstance(a, int)) # Check for an integer type
b = "Hello"
print(isinstance(b, bool)) # Check for a boolean type
c = True
print(isinstance(c, bool)) # Check for a boolean type
```

```
True
False
True
```

ASSIGNING A VALUE TO A VARIABLE

- You can store values in a variable using the “=” sign, which is called the assignment operator.
- There are many different conventions for naming variables but not too long and descriptive is ideal.
- You can check out the Google Python style guide [here](#) .
- Variable names:
 - Start with a letter or underscore.
 - Can only contain alphanumeric [a-Z, 0-9] characters and underscores.
 - Are case sensitive.

```
a = 31.2      # Assign the value 3.12 to the variable called "a"
b = 4         # Assign the value 4 to the variable called "b"
print(a + b)  # Add them up and print the result

part_one = "Good" # Do the same for strings.
part_two = "news"
print(part_one + part_two) # Adding two strings concatenates them
```

35.2
Goodnews

ASSIGNING ONE VARIABLE'S VALUE TO ANOTHER VARIABLE

- You can assign one variable to another:

```
# Recall that the variable 'a' contains the number 31.2  
c = a  
print(c)
```

```
31.2
```

DATA STRUCTURES

CONTAINERS HOLD OBJECTS AND DATA

- So far all the variables we have looked at have had a single value, which is known as a *scalar*.
- But if we are dealing with more than one item, it is useful to collect them into a container (just like a suitcase when you travel).
- Python comes with built-in containers:
 - list
 - dict
 - tuple
 - set
- The choice of data structure depends on the type of task you are performing.
- These core data structures will be supplemented with ones from add-on packages.
- One data structure we will spend a lot of time on later, is called a “DataFrame” from the pandas package.
- And we will see the “ndarray” container from the numpy package, used for holding mathematical objects like vectors and matrices.

LISTS

- A list is a variable length container, within which the elements can be modified (compare to a tuple later).
- It can contain a mix of variable types.
- It can be “nested”, where elements inside the list are themselves other containers.
- List elements can be accessed by position (starting at 0).
- You make a list by enclosing the elements in square brackets [], or by using the “list” function.

```
# Create a list containing a float, integer, string and logical
list_one = [3.14, 10, 'Fantastic', True]

print(type(list_one))
print(list_one)
```

```
<class 'list'>
[3.14, 10, 'Fantastic', True]
```

MANIPULATING LISTS

- To find out the actions you can take on a list, type in `list_one.` and then the tab key.
- You will see: `append`, `clear`, `copy`, `count`, `extend`, `insert`, `index`, `pop`, `remove`, `reverse`, `sort`.
- To find the length of a list, you use the built-in function “`len`”

```
# Finding the length of the list
print(len(list_one))
print("The length of the list is:" + str(len(list_one)))

# Stick a new element on the end of the list:
list_one.append("new element at end")
print(list_one)

# Insert an element at position two (that's the third slot)
list_one.insert(2, "Inserted!")
print(list_one)
```

```
4
The length of the list is:4
[3.14, 10, 'Fantastic', True, 'new element at end']
[3.14, 10, 'Inserted!', 'Fantastic', True, 'new element at end']
```

MORE LIST ACTIVITIES

```
# Remove an element and store it in a variable:  
new_var = list_one.pop(2)  
print(list_one) # Note the element has disappeared  
print(new_var)  # Here it is, stored in a variable.
```

```
[3.14, 10, 'Fantastic', True, 'new element at end']  
Inserted!
```

```
# Combine (concatenate two lists using the "+" operator)  
list_two = [False, 62, 55, "Bobby"]  
print(list_one + list_two)
```

```
[3.14, 10, 'Fantastic', True, 'new element at end', False, 62, 55, 'Bobby']
```

SLICE NOTATION “:”

- To pull out pieces of a list we use the slice notation, which is [start:stop], e.g. [3:6]
- The first possible position is 0, and the slice will get elements up to, but not including stop.

```
# A list with the letters of the alphabet:
alphabet =
    ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

print(len(alphabet)) # A good check!
```

```
26
```

```
print(alphabet[0])    # The first element
print(alphabet[3:6])  # d through f
# Using minus notation lets you work backwards:
print(alphabet[-6:-3]) # The sixth from the end, to the fourth from the end
```

```
a
['d', 'e', 'f']
['u', 'v', 'w']
```

STEP SIZE OF THE SLICE

```
# If you use a second colon you can control the step length  
print(alphabet[::4]) # Every fourth letter from the start
```

```
print(alphabet[::-1]) # Reverse a list
```

```
['a', 'e', 'i', 'm', 'q', 'u', 'y']  
['z', 'y', 'x', 'w', 'v', 'u', 't', 's', 'r', 'q', 'p', 'o', 'n', 'm', 'l', 'k', 'j', 'i', 'h', 'g',  
'f', 'e', 'd', 'c', 'b', 'a']
```

ZIPPING UP TWO LISTS

- If you have two lists you can join up the elements in pairs (tuples).
- These pairs can be put into their own list

```
list_en = ["one", "two", "three"]
list_fr = ["un", "deux", "troi"]

translate = list(zip(list_en, list_fr)) # zip the lists together
print(translate) # Print the whole list:
print(translate[2]) # Just the element in the third position
```

```
[('one', 'un'), ('two', 'deux'), ('three', 'troi')]
('three', 'troi')
```

SORTING A LIST

- Lists have a “sort” method and you can also control which metric, elements are sorted by.
- This is an “in-place” sort, as the original list is itself changed. It does not create a new copy.

```
# My family
family = ["Richard", "Lisa", "David", "Thomas", "Alex"]
family.sort() # A plain sort
print(family)

family.sort(key = len) # Sort based on the length of the name
print(family)
```

```
['Alex', 'David', 'Lisa', 'Richard', 'Thomas']
['Alex', 'Lisa', 'David', 'Thomas', 'Richard']
```


THE DICT DATA STRUCTURE

- The dict data structure is a very useful container.
- A dict contains “key-value” pairs.
- You don’t access a dict by position (unlike a list), but rather by the name of the key.
- This can be much more efficient for looking up values, rather than maintaining and searching lists.
- To create a dict directly use curly braces {}.

```
# A "dict" structure with a key (state abbreviation) and value: a list with full state name and capital city.  
  
states = {'PA': ["Pennsylvania", "Harrisburg"],  
          'MD': ["Maryland", "Annapolis"],  
          'FL': ["Florida", "Tallahassee"]}  
print(states)
```

```
{'PA': ['Pennsylvania', 'Harrisburg'], 'MD': ['Maryland', 'Annapolis'], 'FL': ['Florida',  
'Tallahassee']}
```

ACCESSING THE ELEMENTS OF A DICT

- By using the key name, you can get at the value.
- To see if a key is in the dict, use the “in” command, which returns a logical value.

```
print(states['MD']) # Returns the relevant list.  
print(states['MD'][1]) # Drills down on the returned list.  
'CA' in states # looks to see if one of the keys is called 'CA'
```

```
['Maryland', 'Annapolis']  
Annapolis
```

```
False
```

ADDING AND REMOVING ELEMENTS TO THE ‘DICT’

- Using a key that does not yet exist, automatically creates a new key-value.
- The `del` keyword and `pop` method, delete and extract values.

```
# Add a new element
states['OR'] = ["Oregon", "Eugene"]
print(states)
del states['MD'] # Delete the 'MD' element using the "del" keyword.
print(states)
states.pop('PA') # Remove and return the PA element
```

```
{'PA': ['Pennsylvania', 'Harrisburg'], 'MD': ['Maryland', 'Annapolis'], 'FL': ['Florida',
'Tallahassee'], 'OR': ['Oregon', 'Eugene']}
{'PA': ['Pennsylvania', 'Harrisburg'], 'FL': ['Florida', 'Tallahassee'], 'OR': ['Oregon', 'Eugene']}
```

```
['Pennsylvania', 'Harrisburg']
```

GETTING ALL THE KEYS OR VALUES IN A 'DICT'

- To get at all the keys in a dict use the `.keys()` method.
- To get at all the values in a dict, use the `.values()` method.

```
# The keys of the dict, contained in a list.  
print(list(states.keys()))
```

```
# The values of the dict, contained in a list.  
print(list(states.values()))
```

```
['FL', 'OR']  
[['Florida', 'Tallahassee'], ['Oregon', 'Eugene']]
```

EDITING A VALUE IN THE DICT

- The capital of Oregon is ‘Salem’, not ‘Eugene’.
- We can drill down into the dict and assign it the new value:

```
# The key is 'OR', and the capital city is in the second slot in the list:  
states['OR'][1] = 'Salem'  
print(states)
```

```
{'FL': ['Florida', 'Tallahassee'], 'OR': ['Oregon', 'Salem']}
```

TUPLES

- tuples are quite similar to lists, but they are “fixed”.
- You can’t change the number of elements in the tuple.
- You can’t change the objects stored in each slot.
- The formal term is that they are “immutable”.
- You can create them tuples just by using a comma separated set of values, or use the parentheses() for readability.

```
# A tuple with four elements
tuple_one = 2.1, 'pi', "hello", False
print(tuple_one)

# Accessing an element(2) by position:
print(tuple_one[1:3])
print(tuple_one[-2:-1]) # The penultimate element.
print(len(tuple_one)) # The number of elements in the tuple.
```

```
(2.1, 'pi', 'hello', False)
('pi', 'hello')
('hello',)
4
```

ASSIGNMENT OF TUPLES

- You can take the elements of a tuple and store them in variables.

```
a, b, c, d = tuple_one # Unpack the tuple to variables
print("The variable 'b' contains the value: " + b)
e, f = tuple_one # If there aren't enough variables the assignment fails.
```

```
The variable 'b' contains the value: pi
```

```
-----

ValueError                                Traceback (most recent call last)

<ipython-input-26-5189f0163da0> in <module>
      1 a, b, c, d = tuple_one # Unpack the tuple to variables
      2 print("The variable 'b' contains the value: " + b)
----> 3 e, f = tuple_one # If there aren't enough variables the assignment fails.

ValueError: too many values to unpack (expected 2)
```

COLLECTING THE TUPLE ASSIGNMENT LEFT-OVERS

```
# The below syntax will collect all remaining elements into a catch-all variable.  
# Note the "*" in front of the left_overs variable, which means it can accept any number of variables.  
e, f, *left_overs = tuple_one  
print(left_overs)
```

```
['hello', False]
```


THE SET CONTAINER

- The set container has *no ordering* (so you can't access it by position), and its elements are unique.
- Which is the same as a *set* in mathematics.
- Like a dict, they can be created using the curly brackets {}.
- They can also be created by using the set function.
- The set is mutable, but its elements must be immutable.

```
# Make a first set
set_one = {'a', 'd', 'd', 'yes', True, 3.13, 'd'}
print(set_one) # Note how duplicate values are discarded.
# And now a second one
set_two = set(['fifteen', 'yes', 42.3, 42.3, 3.13, 'fifteen', 'a'])
print(set_two)
```

```
{True, 3.13, 'a', 'd', 'yes'}
{3.13, 'a', 42.3, 'fifteen', 'yes'}
```

OPERATIONS ON SETS

- The two main operations on sets are to find their union and their intersection.
- The union comprises all elements in the two sets.
- The intersection are those elements that appear in both sets.
- You can also find the number of elements in a set using the len command again.
- To see all the operations on sets use tab completion, type in set_one. and then hit the tab key.
- Another useful set method is “difference”, which shows you which elements are in one set, but not the other.

```
print(set_one.union(set_two))      # The union of two sets
print(set_one.intersection(set_two)) # The intersection of the two sets
print(len(set_one))                # The number of elements in set_one
```

```
{True, 3.13, 'a', 42.3, 'fifteen', 'd', 'yes'}
{'yes', 3.13, 'a'}
5
```

AN EXAMPLE USING SETS

```
class_list = {'Mary', 'Bridget', 'Bill', 'Andreas', 'Matt', 'Cecilia'}  
test_takers = {'Bill', 'Bridget', 'Andreas', 'Bruno'}  
# 1. Who is in the class and took the test?  
# 2. Who is in the class, but didn't take the test?  
# 3. Who took the test, but isn't registered for the class?  
  
print (class_list.intersection(test_takers))  
print (class_list.difference(test_takers))  
print (test_takers.difference(class_list))
```

```
{'Bridget', 'Bill', 'Andreas'}  
{'Mary', 'Cecilia', 'Matt'}  
{'Bruno'}
```

WHICH CONTAINER SHOULD YOU USE?

- This will be very dependent on the objective of the problem.
- A dictionary is going to be useful when you are frequently looking up values by name.
- A list will be useful, when there is an ordered type structure to the underlying data and you are looking things up by position.

SUMMARY

SUMMARY

- Got to know the Spyder IDE.
- Used the console for interactive Python activities.
- Aware of various data types.
- Introduced to four built-in Python data structures.
- Accessed lists and dicts through *slicing* .

NEXT TIME

NEXT TIME

- More Python essentials:
 - Control flow
 - Branching
 - Iteration
 - List comprehensions
- Jupyter notebooks
- Markdown