

# Python

Bootcamp 2021

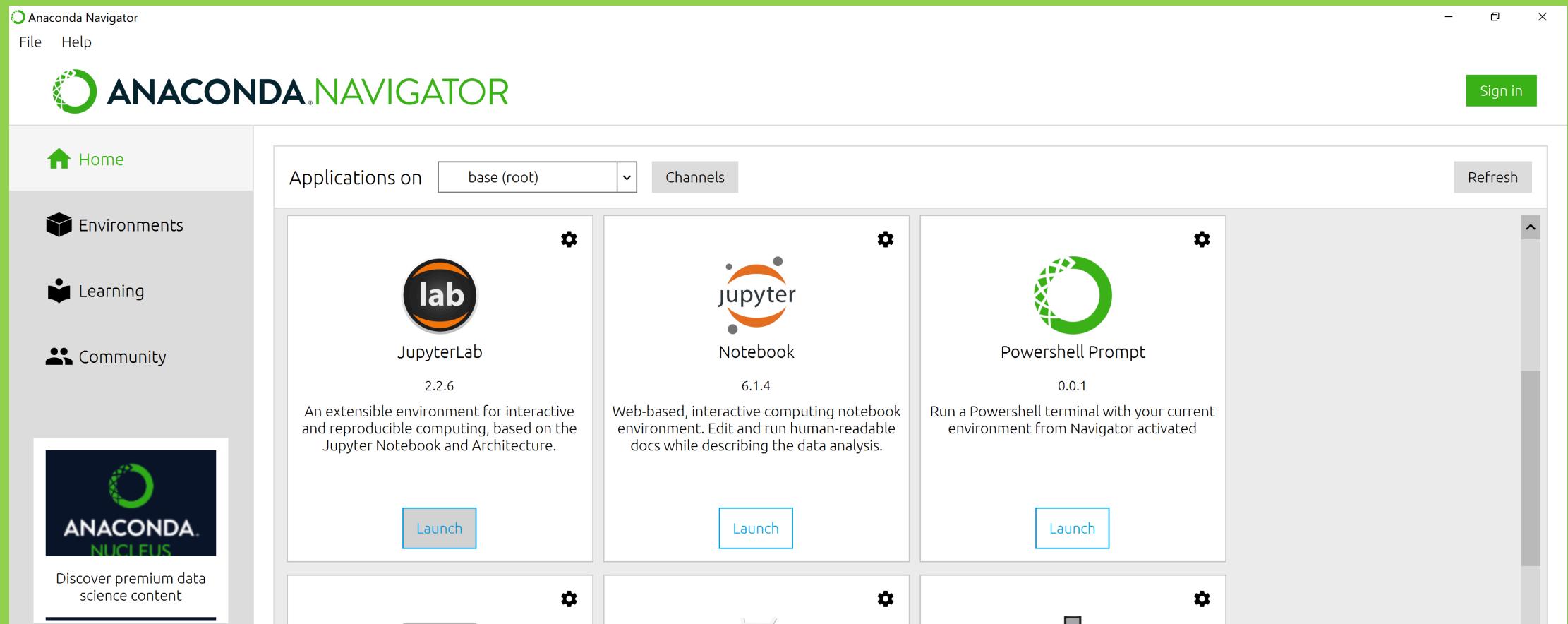
# Outline for today

- Setup Anaconda
- Running Python
- Variables and Assignment
- Data Type
- Built-in functions
- Conditionals
- Loops

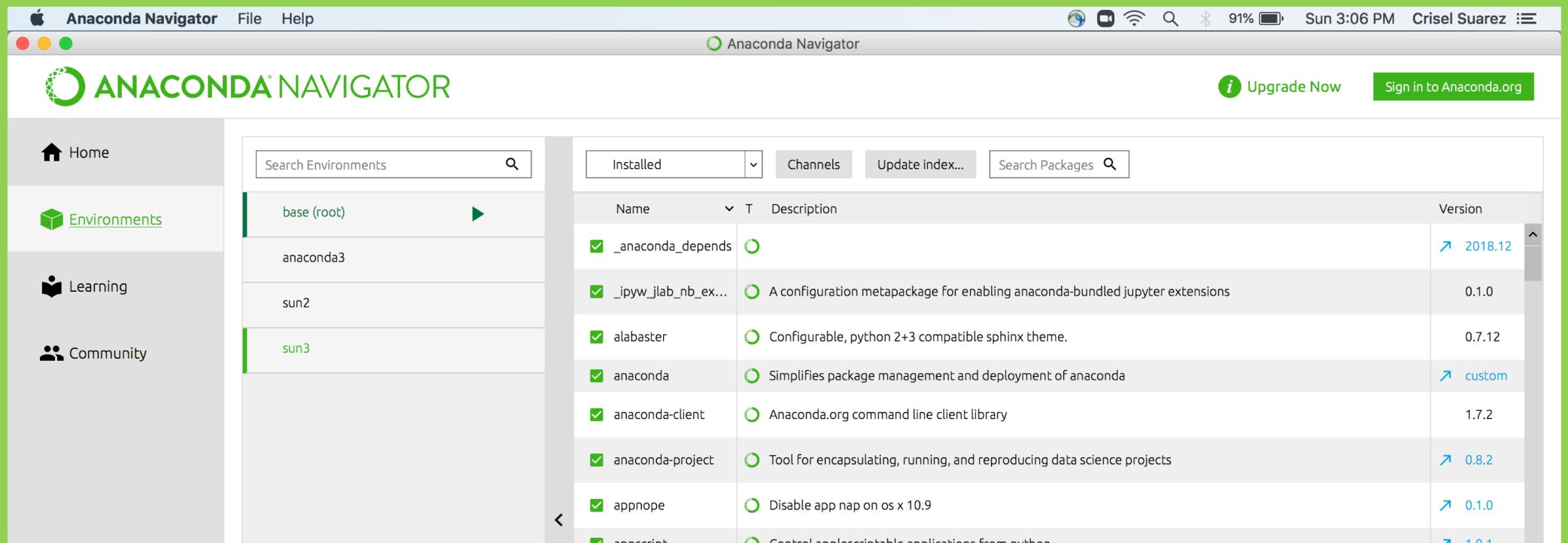
# Outline for today

- Setup Anaconda ←
- Running Python
- Variables and Assignment
- Data Type
- Built-in functions
- Conditionals
- Loops

# Setup Anaconda



# Setup Anaconda



# Setup Anaconda

- >On your bash shell
- \$ conda create --name bootcamp2021
- proceed ([y]/n)?
- Y
- \$ conda info --envs
- \$ conda env list
- \$ conda activate bootcamp2021
- \$ conda list -n bootcamp2021
- \$ conda install package-name
- \$ conda install package-name=2.3.4
- <https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>
- <https://docs.anaconda.com/anaconda/user-guide/tasks/install-packages/>
- \$ conda create --name bootcamp2021 --clone base

# Scripts /Spyder/Jupyter Notebook/JupyterLab

- All have pros/cons
- Choose what works best for you
- It is okay to switch between platforms

# Python Scripts

- Run scripts on your bash shell

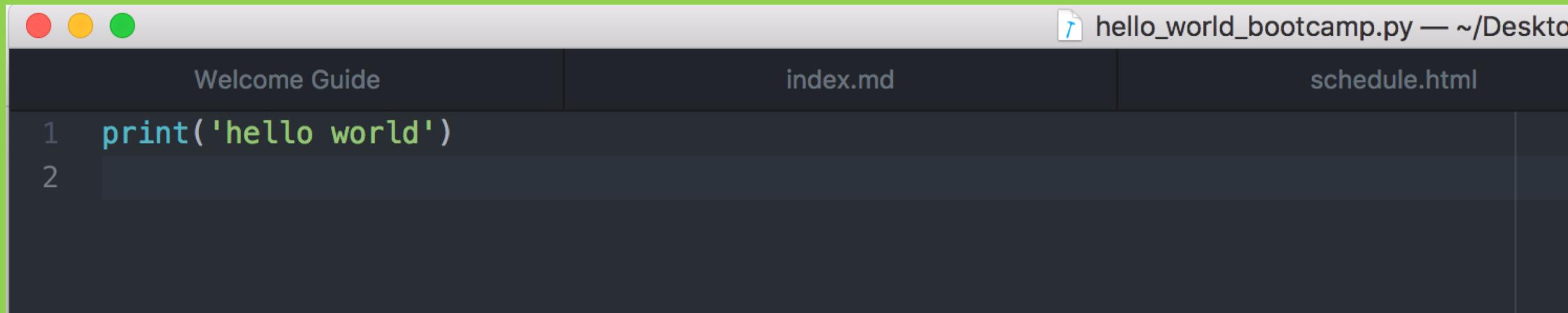
- \$python

```
>>>  
>>>print('hello world')  
>>>exit() #Go back to your bash shell ($)
```

- \$ vim hello\_world.py
- print('hello world')
- \$python hello\_world.py

- vim
  - Insert mode (i)
  - Type your script/notes
  - esc
  - :wq

# Python Scripts-Atom/Text Editor



The screenshot shows the Atom text editor interface. The title bar displays the file name "hello\_world\_bootcamp.py" and the path "~Desktop". The editor has a dark theme with tabs for "Welcome Guide", "index.md", and "schedule.html". The main code editor area contains two lines of Python code:

```
1 print('hello world')
2
```

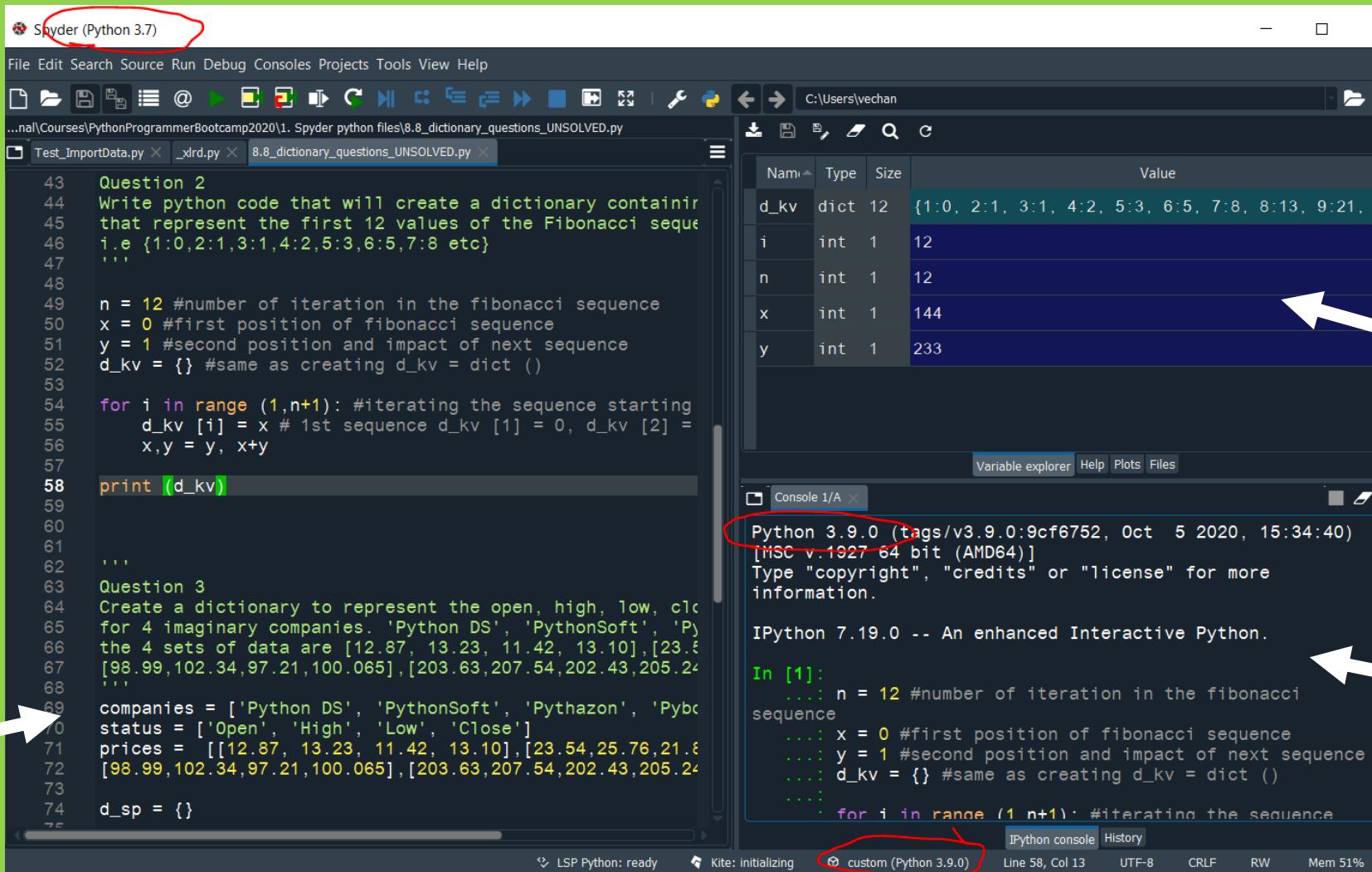
On your bash shell

```
$python hello_world_bootcamp.py
```

```
hello world
```

# Spyder

Script  
Code  
goes here



# Declared Variables

# Output

# Spyder

Run your code

The screenshot shows the Spyder Python 3.7 IDE interface. A red circle highlights the title bar "Spyder (Python 3.7)". A white arrow points from the "Run your code" text to the toolbar icon for running the current file. The code editor displays a script named "8.8\_dictionary\_questions\_UNSOLVED.py" containing two questions and their solutions. The first question creates a Fibonacci sequence dictionary. The second question creates a dictionary for stock prices across four companies. The variable explorer shows the state of the variables:

Name	Type	Size	Value
d_kv	dict	12	{1:0, 2:1, 3:1, 4:2, 5:3, 6:5, 7:8, 8:13, 9:21, 10:34, 11:55, 12:89}
i	int	1	12
n	int	1	12
x	int	1	144
y	int	1	233

The IPython console output shows the Python version and the code being run:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40)
[MSC v.1927 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more
information.

IPython 7.19.0 -- An enhanced Interactive Python.

In [1]:
...: n = 12 #number of iteration in the fibonacci
sequence
...: x = 0 #first position of fibonacci sequence
...: y = 1 #second position and impact of next sequence
...: d_kv = {} #same as creating d_kv = dict ()
...:
...: for i in range (1, n+1): #iterating the sequence
```

A red circle highlights the "custom (Python 3.9.0)" tab in the IPython console tab bar.

# Spyder

Debug your code

The screenshot shows the Spyder Python IDE interface. A red circle highlights the title bar "Spyder (Python 3.7)". A large white arrow points from the text "Debug your code" to the toolbar, which contains various icons for file operations, run, debug, and help. The main code editor window displays a script named "8.8\_dictionary\_questions\_UNSOLVED.py". The code implements a Fibonacci sequence and prints it as a dictionary. A red box highlights the status bar at the bottom, which shows "custom (Python 3.9.0)".

File Edit Search Source Run Debug Console Projects Tools View Help

...nal\Courses\PythonProgrammerBootcamp2020\1. Spyder python files\8.8\_dictionary\_questions\_UNSOLVED.py

Test\_ImportData.py xrd.py 8.8\_dictionary\_questions\_UNSOLVED.py

```
43 Question 2
44 Write python code that will create a dictionary containing
45 that represent the first 12 values of the Fibonacci sequence
46 i.e., {1:0,2:1,3:1,4:2,5:3,6:5,7:8 etc}
47
48 n = 12 #number of iteration in the fibonacci sequence
49 x = 0 #first position of fibonacci sequence
50 y = 1 #second position and impact of next sequence
51 d_kv = {} #same as creating d_kv = dict ()
52
53 for i in range (1,n+1): #iterating the sequence starting
54     d_kv [i] = x # 1st sequence d_kv [1] = 0, d_kv [2] =
55     x,y = y, x+y
56
57 print (d_kv)
58
59
60
61
62 ...
63 Question 3
64 Create a dictionary to represent the open, high, low, close
65 for 4 imaginary companies. 'Python DS', 'PythonSoft', 'Py
66 the 4 sets of data are [12.87, 13.23, 11.42, 13.10],[23.5
67 [98.99,102.34,97.21,100.065],[203.63,207.54,202.43,205.24
68 ...
69 companies = ['Python DS', 'PythonSoft', 'Pythazon', 'Pyb
70 status = ['Open', 'High', 'Low', 'Close']
71 prices = [[12.87, 13.23, 11.42, 13.10],[23.54,25.76,21.8
72 [98.99,102.34,97.21,100.065],[203.63,207.54,202.43,205.24
73
74 d_sp = {}
```

Variable explorer Help Plots Files

Console 1/A

Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40)  
[MSC v.1927 64 bit (AMD64)]  
Type "copyright", "credits" or "license" for more  
information.

IPython 7.19.0 -- An enhanced Interactive Python.

In [1]:

```
...: n = 12 #number of iteration in the fibonacci
sequence
...: x = 0 #first position of fibonacci sequence
...: y = 1 #second position and impact of next sequence
...: d_kv = {} #same as creating d_kv = dict ()
...:
...: for i in range (1,n+1): #iterating the sequence
```

IPython console History

LSP Python: ready Kite: initializing custom (Python 3.9.0) Line 58, Col 13 UTF-8 CRLF RW Mem 51%

More on this later

# Jupyter Lab (.ipynb)

\$ jupyter lab

The screenshot shows the Jupyter Lab interface. On the left is a sidebar with tabs for Files, Running, Commands, Cell Tools, and Tabs. The Files tab shows a list of notebooks: Data.ipynb (selected), Fasta.ipynb, Julia.ipynb, and R.ipynb. The Running tab shows a list of kernels: Terminal 1, README.md, Data.ipynb (Python 3), Julia.ipynb, Fasta.ipynb, and postBuild. The main area has two code cells. The first cell (In [17]) contains Python code to import pandas and read a CSV file, with its output (Out[17]) showing the first five rows of the Iris dataset. The second cell (In [20]) contains Python code to display a GeoJSON map of Washington D.C. with various locations marked by blue pins. Two black arrows point from the text "Cell - Code" and "Output" to the respective parts of the screenshot.

```
In [17]: import pandas
df = pandas.read_csv('../data/iris.csv')
df.head(5)
```

```
Out[17]: sepal_length  sepal_width  petal_length  petal_width  species
0           5.1         3.5          1.4         0.2     setosa
1           4.9         3.0          1.4         0.2     setosa
2           4.7         3.2          1.3         0.2     setosa
3           4.6         3.1          1.5         0.2     setosa
4           5.0         3.6          1.4         0.2     setosa
```

```
In [20]: from IPython.display import GeoJSON
GeoJSON(s, layer_options={'minZoom': 11})
```

Cell – Code

Output

To run a cell:  
shift + enter

<https://jupyterlab.readthedocs.io/en/stable/user/notebook.html>

# Jupyter notebook (.ipynb)

\$ jupyter notebook

- When in Command mode (esc/gray),

- The b key will make a new cell below the currently selected cell.
- The a key will make one above.
- The x key will delete the current cell.
- The z key will undo your last cell operation (which could be a deletion, creation, etc).

# Jupyter notebook (.ipynb)

- Markdown great for commenting/adding notes to your code!
- A simple plain-text format for writing lists, links, and other things that might go into a web page.

Turn the current cell into a **Markdown cell** by entering the Command mode (**Esc**) and press the **M** key.

In [ ]: will disappear to show it is no longer a code cell and you will be able to write in Markdown.

Turn the current cell into a **Code cell** by entering the Command mode (**Esc**) and press the **y** key

# Markdown – html

- \* Use asterisks
- \* to create
- \* bullet lists.



Lists

# A Level-1 Heading



Headings

## A Level-2 Heading (etc.)

[Create links](http://software-carpentry.org) with ` [...] (...)`.



urls + links

Or use [named links][data\_carpentry]. [data\_carpentry]:  
http://datacarpentry.org

# Outline for today

- Setup Anaconda
- Running Python
- Variables and Assignment 
- Data Type
- Built-in functions
- Packages
- Conditionals
- Loops

# Variables and Assignments

- In Python the = symbol assigns the value on the right to the name on the left
- age = 42
- my\_name = 'Crisel Suarez'
- Grade1 = 'A'
- Variable names
  - can **only** contain letters, digits, and underscore \_
  - cannot start with a digit
  - are **case sensitive** (age, Age and AGE are three different variables)

# Variables and Assignments

- `first_name = 'Kathy'`
  - `age = 10`
  - `print(first_name, 'is', age, 'years old')`
  - Variables can be used in calculations:
    - `new_age = age +10`
  - Indexing
    - `print(first_name[0])`
- \*\*\* Python indexing starts at 0 \*\*\*

# Outline Wednesday



- Jupyter Magic Commands
  - Indexing and Slices
  - Lists
  - Built-in Functions
  - Conditionals
  - Loops
  - Functions
- 

# Key Points

- Use variables to store values.
- Use `print()` to display values.
- Variables persist between cells.
- Variables must be created before they are used.
- Variables can be used in calculations.

# Jupyter Magic Commands

- %run hello.py
- %%time
- % who
- %who str | % who int
- %pinfo <variable>
- %env
- %matplotlib inline
- %load hello.py
- %lsmagic



[https://www.tutorialspoint.com/jupyter/ipython\\_magic\\_commands.htm](https://www.tutorialspoint.com/jupyter/ipython_magic_commands.htm)

# Jupyter Magic Commands

- Can run Unix commands straight from your Jupyter Notebooks
- !
- !head -n 5 haiku.txt
- !pip install astropy
- Almost all the things we learned in Unix we can use in Jupyter Notebooks

# Data Types

- str() – String
  - Concatenation +
  - Repetition \*
- int()- integer
- Float() - decimals
- Type() > What kind of data type

# Math

- Add +
- Subtract -
- Multiply \*
- Divide /
- Power \*\*
- Reminder %
- Absolute value abs()

# Operators

- Equal to ==
- Not equal to !=
- Greater than >
- Less than <
- Greater or equal >=
- Less or equal <=

# Operators

- and
- or
- in (Membership)
- not in (Membership)
- True
- False

# Outline Wednesday

- Jupyter Magic Commands
- Indexing and Slices ←
- Lists
- Built-in Functions
- Conditionals
- Loops
- Functions

# Indexing and Slices

- [start:stop]
- atom\_name = 'sodium'
- print(atom\_name[0:3])
  - > sod
- len(atom\_name)
- 6

# Outline Wednesday

- Jupyter Magic Commands
- Indexing and Slices
- Lists 
- Built-in Functions
- Conditionals
- Loops
- Functions

# Lists

- Storing multiple variables
- pressures = [0.273, 0.275, 0.277, 0.275, 0.276]
- **print('pressures:', pressures)**
- **print('length:', len(pressures))**
- **print('zeroth item of pressures:', pressures[0])**
- pressures[0] = 0.265
-

# Lists – Appending

- `list_name.append()`
- `primes = [2, 3, 5]`
- `print('primes is initially:', primes)`
- `primes.append(7)`
- `print('primes has become:', primes)`

# Lists – Deleting

- `del list_name[index]` to remove an element from a list
- `primes = [2, 3, 5, 7, 9]`
- `print('primes before removing last item:', primes)`
- `del primes[4]`
- `print('primes after removing last item:', primes)`

# List- Empty []

- Empty\_list = []
- Helpful as a starting point for collecting values

# Practice:

- `print('string to list:', list('tin'))`
- `print('list to string:', ".join(['g', 'o', 'l', 'd']))`

What does `list` do?

What does `.join` do?

\*We will come back to `list` with Numpy's version ...arrays

# Key Points

- Use an index to get a single character from a string.
- Use a slice to get a substring.
- Use the built-in function `len()` to find the length of a string.
- Python is case-sensitive.
- Use meaningful variable names

# Dictionaries {} or dict()

- Mutable key-value pairs
- `zoo = {'cats' : 4 , 'dogs': 5, 'goats': 3, 'camels' : 2 }`
- `person = dict(name = "John", age = 36, country = "Norway")`
- `zoo['cats']`
  - `> 4`
- `zoo.keys()`
- `zoo.values()`
- `zoo.items()`

# Dictionaries

- food = {'breakfast' : 2 , 'lunch': 'salad',  
'dinner': {'first\_course' : 'soup',  
'second\_course': 'chicken' }  
desert = ['flan', 'coockies', 'NY\_chessecakes']}
- food['dinner']['first\_course']
- food['dessert'][0]

# Tuple – ()

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with parentheses ()
- Allows duplicated items

```
thistuple = ("apple", "banana", "cherry")
```

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
```

# Sets – {}

- Unordered
- Unchangeable
- No duplicate values.

```
thistuple = {"apple", "banana", "cherry"}
```

```
thistuple = {"apple", "banana", "cherry", "apple", "cherry"}
```

# Python Collections

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered\* and changeable. No duplicate members.

# Outline Wednesday

- Jupyter Magic Commands
- Indexing and Slices
- Lists
- Built-in Functions 
- Conditionals
- Loops
- Functions

# Built-in functions

- Think math function
- $f(x) = x + 5$
- $x \rightarrow$  input
- $f(x) \rightarrow$  output
- Functions can take 0 or many arguments
- `print()`
- $f(x_1, x_2, x_3, \dots) = x_1 + x_2 + x_3 + \dots$

# Built-in functions

- `max(1,2,3)`
- `min(5,6,7)`
- `round(3.712, 1)` #rounds to 1 decimal place
  
- `help(round)`

# Functions attached to objects are called methods

- Methods have parentheses like functions, but come after the variable.

```
my_string = 'Hello world!' # creation of a string object
```

```
print(my_string.swapcase())
```

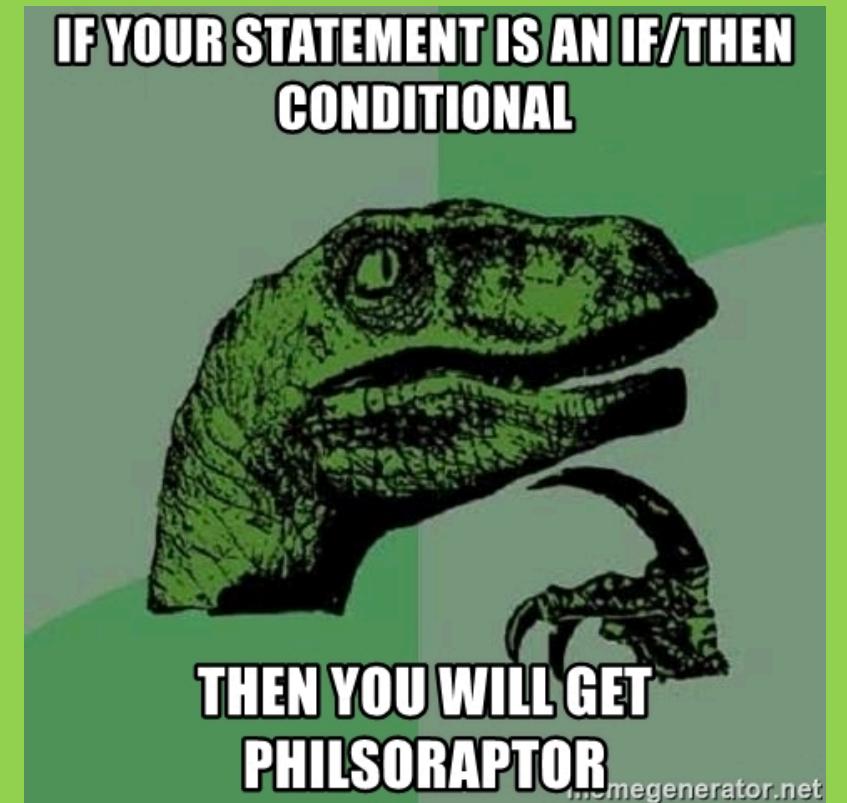
```
# calling the swapcase method on the my_string object
```

# Outline Wednesday

- Jupyter Magic Commands
- Indexing and Slices
- Lists
- Built-in Functions
- Conditionals 
- Loops
- Functions

# Conditionals

- if (condition is True):  
    then do something
- if (condition is True):  
    then do something
- else:
  - Do something else
- if (condition is True):  
    then do something
- elif (this condition is true):
  - then do this
- else:
  - Do this



# Conditionals – Try it out

- mass = 3.4
- If mass > 3.0:
  - print('Mass is ', mass)
- if mass > 3:
  - print('Mass is less than 3')
- else:
  - print('Mass is more than 3')
- if mass < 3.7:
  - print('mass less than 3.7')
- elif (if mass > 3.2 ):
  - print('mass greater than 3.2')
- else:
  - print(mass greater than 3.7 or less than 3.2)

# Conditionals – Try it out

- mass = 3.4
- If ((mass < 3.7) and (mass >3.2)):
  - print(mass less than 3.7 or greater than 3.2)
- mass = 3.4
- If ((mass < 3.7) or (mass >3.2)):
  - print(mass less than 3.7 or greater than 3.2)
- mass = 3.8
- If ((mass < 3.7) and (mass >3.2)):
  - print(mass less than 3.7 or greater than 3.2)
- mass = 3.8
- If ((mass < 3.7) or (mass >3.2)):
  - print(mass less than 3.7 or greater than 3.2)

# Conditionals

p	q	p and q
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

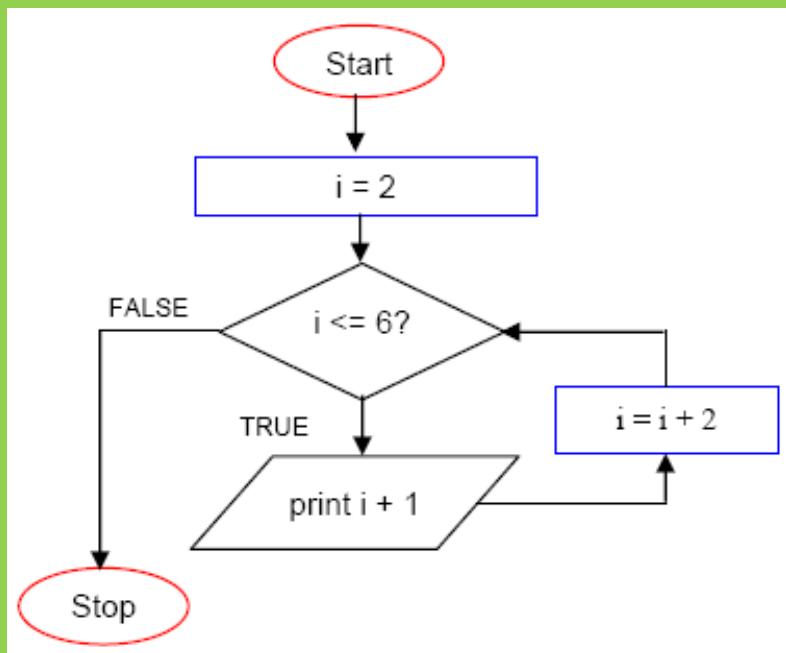
p	q	p or q
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

# Outline Wednesday

- Jupyter Magic Commands
- Indexing and Slices
- Lists
- Built-in Functions
- Conditionals
- Loops 
- Functions

# Loops

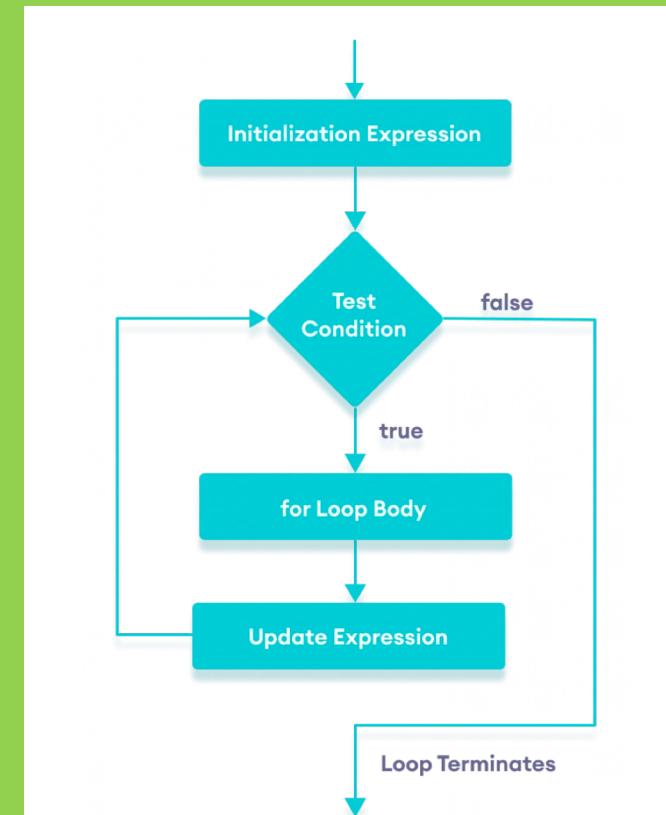
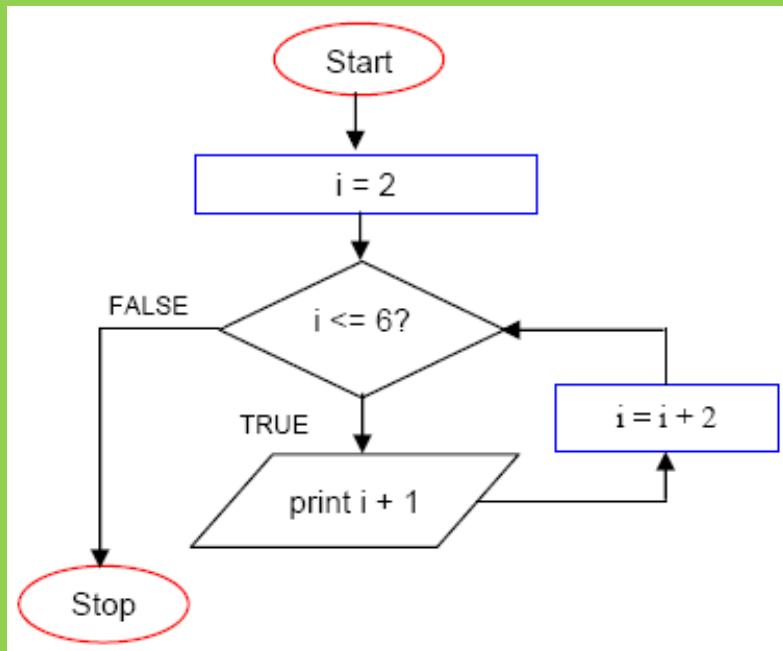
**Loops** are a programming construct which allow us to repeat a command or set of commands for each item in a list. As such they are key to productivity improvements through automation



i	i <= 6	Output
2	True	3
4	True	5
6	True	7
8	False	

# Loops

**Loops** are a programming construct which allow us to repeat a command or set of commands for each item in a list. As such they are key to productivity improvements through automation



# Loops

- **for** number **in** [2, 3, 5]:
  - **print**(number)
- primes = [2, 3, 5]
- **for** p **in** primes:
  - squared = p \*\* 2
  - cubed = p \*\* 3
  - **print**(p, squared, cubed)

# Loops

- The built-in function `range` produces a sequence of numbers.
- Not a list: the numbers are produced on demand to make looping over large ranges more efficient.
- `print('a range is not a list: range(0, 3)')`
- `for number in range(0, 3):`
  - `print(number)`

# Loops – Practice

- # List of word lengths: `["red", "green", "blue"] => [3, 5, 4]`
- `lengths = _____`
- **for** word **in** `["red", "green", "blue"]:`
  - `lengths._____(___)`
- **print**(`lengths`)

# Loops – Practice

- # List of word lengths: `["red", "green", "blue"] => [3, 5, 4]`
- `lengths = []`
- **for** word **in** `["red", "green", "blue"]:`
  - `lengths.append(len(word))`
- **print**(`lengths`)

# Loops – Practice

- # Concatenate all words: `["red", "green", "blue"] => "redgreenblue"`
- `words = ["red", "green", "blue"]`
- `result = _____`
- `for _____ in _____:`
  - `_____`
- `print(result)`

# Loops – Practice

- # Concatenate all words: `["red", "green", "blue"] => "redgreenblue"`
- `words = ["red", "green", "blue"]`
- `result = ""`
- **for** word **in** words:
  - `result = result+word`
- **print(result)**

# Practice

- Write a program that prints the following pattern:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

# Practice

- Write a program that prints the following pattern:

```
*  
**  
***  
****  
*****
```

```
for star in range(7):  
    print('*' * star)
```

# While Loops

- Need to define an indexing variable\*\*\*

```
i = 1
```

```
while i < 6:
```

```
    print(i)
```

```
    i += 1
```

```
i = 1
```

```
while i < 6:
```

```
    print(i)
```

```
    i += 1
```

```
else:
```

```
    print("i is no longer less than 6")
```

**\*\*\*Loop can run forever\*\*\***

# Conditionals + Loops

```
i = 0  
while i < 6:  
    i += 1  
    if i == 3:  
        print("i is 3")  
print(i)  
  
masses = [3.54, 2.07, 9.22, 1.86, 1.71]  
for m in masses:  
    if m > 3.0:  
        print(m, 'is large')  
    else:  
        print(m, 'is small')
```

# Loops

- **continue** - stop the current iteration, and continue with the next

```
fruits =  
["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        continue  
    print(x)
```

```
i = 0  
while i < 6:  
    i += 1  
    if i == 3:  
        continue  
    print(i)
```

# Loops

- **break** - stop the loop even if the while condition is true

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

# Loops

- **pass** – “Empty loop”

```
for x in [0, 1, 2]:  
    pass
```

# Nested Loops

```
persons = [ "John", "Marissa", "Pete", "Dayton" ]
restaurants = [ "Japanese", "American", "Mexican",
"French" ]

for person in persons:
    for restaurant in restaurants:
        print(person + " eats " + restaurant)
```

# Nested Conditionals

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

# Keypoints

- Use `if` statements to control whether or not a block of code is executed.
- Conditionals are often used inside loops.
- Use `else` to execute a block of code when an `if` condition is *not* true.
- Use `elif` to specify additional tests.
- Create a table showing variables' values to trace a program's execution.

# Keypoints

- A *for loop* executes commands once for each value in a collection.
- A for loop is made up of a collection, a loop variable, and a body.
- The first line of the for loop must end with a colon, and the body must be indented.
- Indentation is always meaningful in Python.
- Make meaningful loop variables
- The body of a loop can contain many statements.
- Use range to iterate over a sequence of numbers.

# Outline Wednesday

- Jupyter Magic Commands
- Indexing and Slices
- Lists
- Built-in Functions
- Conditionals
- Loops
- Functions ←

# Functions def()

\*\*\*Functions return something

```
def print_greeting():
    print('Hello!')
```

```
def print_date(year, month, day):
    joined = str(year) + '/' + str(month) + '/' + str(day)
    print(joined)
```

```
def average(values):
    if len(values) == 0:
        return None
    return sum(values) / len(values)
```

# Practice

- Fill in the blanks to create a function that takes a list of numbers as an argument and returns the first negative value in the list. What does your function do if the list is empty?

```
def first_negative(values):
    for v in ____:
        if ____:
            return ____
```

# Practice

- Fill in the blanks to create a function that takes a list of numbers as an argument and returns the first negative value in the list. What does your function do if the list is empty?

```
def first_negative(values):
    for v in values:
        if v<0:
            return v
```

# Functions + Variable Scope

- ***Global variable***

- Defined outside any particular function.
- Visible everywhere.

- ***Local variable***

- Defined inside the function.
- Not visible in the main program.

pressure = 103.9

def adjust(t):

    temperature = t \* 1.43 / pressure

    return temperature

# Keypoints

- Break programs down into functions to make them easier to understand.
- Define a function using `def` with a name, parameters, and a block of code.
- Defining a function does not run it.
- Arguments in call are matched to parameters in definition.