Introduction to Python

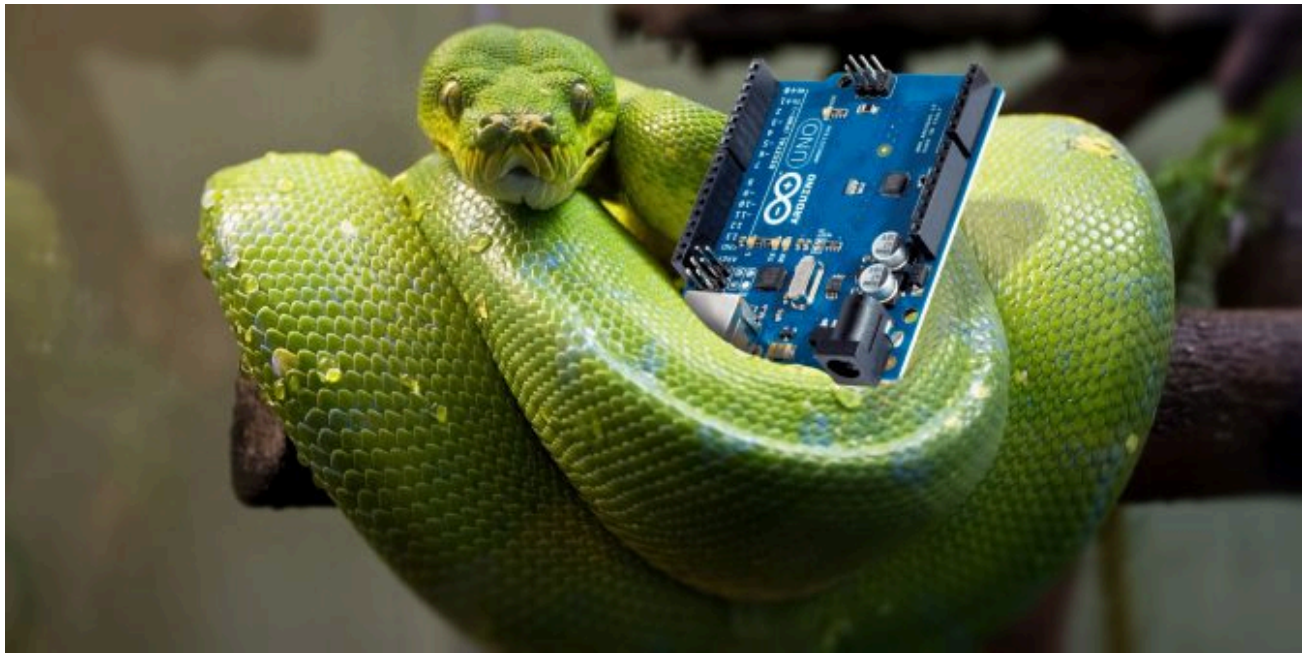
Melody Y. Huang

m.huang@ucla.edu

UCLA Masters of Applied Economics Bootcamp

Motivation

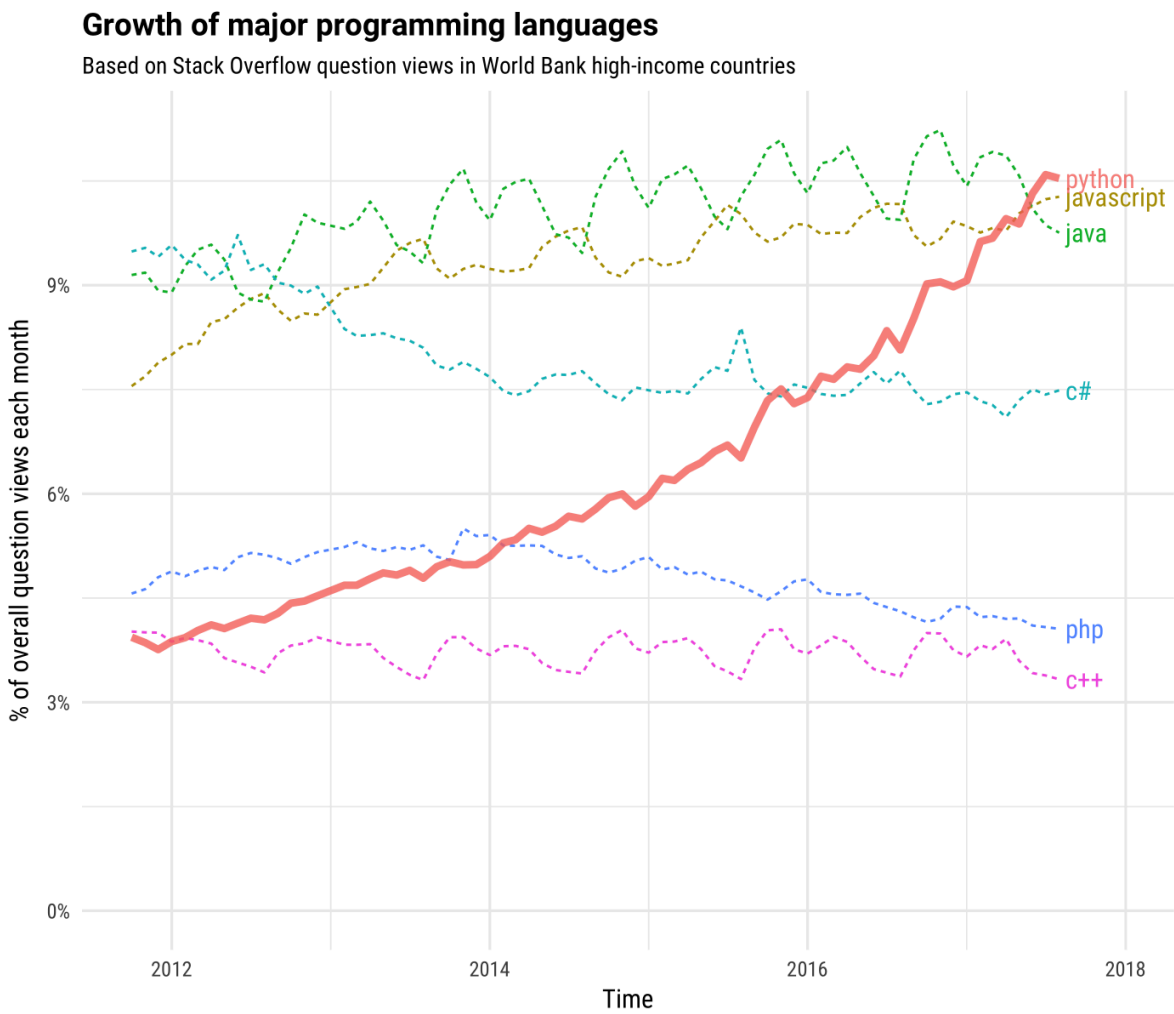
- What is Python?



Motivation (cont.)

- Why learn Python?
  - Data Science
  - Machine Learning

Motivation (cont.)



Installation

- Recommended:
  - anaconda distribution
    - Will provide access to popular IDE's like Spyder and Jupyter Notebook, and will auto-install many popular packages
- Alternatives:
  - Google Colab
  - Terminal/Console (in line, or as a script)
  - PyCharm
  - iPython

Installation (cont.)

- Other helpful things to install:
  - brew 🍺 (Link: https://brew.sh/)
  - pip (`pip install [package_name]`)
    - To install pip: `easy_install pip`

Packages in Python

- Packages (or libraries) in Python provide pre-written functions and objects
- If we want to use a function from a specific library, we have to import that specific library into our workspace

Packages in Python (cont.)

- Functions within the library can be accessed by calling the library name, followed by a period, and then the function name:

```
#Import library
import [library_name]
#Access function
[library_name].[function_name]
```

Packages in Python (cont.)

- If you just want to import a single function:

```
from [library_name] import [function_name]
```

  - If you do that, you can simply call the function name.

- You can even rename functions:

```
from [library_name] import [function_name]
as MyFunction
```

Packages in Python (cont.)

- If you want to import an entire library, you can do so by:

```
from [library_name] import *
```

- This is usually considered bad practice due to naming conflicts.
- **Don't do this!**

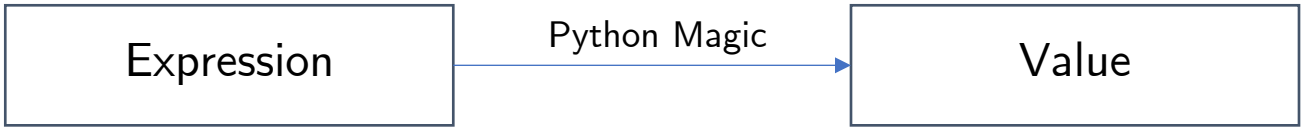
Expressions

- Python is really a fancy calculator:

```
3+4
```

```
7
```

- When we type stuff into the console, this is known as an *expression*.
- Expressions are evaluated by Python to return a *value*.



Expressions (cont.)

- We can use *variables* to help hold expressions.

```
x = 3+4
print(x)
```

```
7
```

- When we tell Python to do something explicitly (i.e., print, create a variable), we call this a *statement*.

Data Types

- Expressions can take on different data types.
- Common data types:
  - Numerics (numbers)
  - Strings (characters)
  - Boolean (true/false)
  - Lists

Data Types (cont.)

- Different data types will have various attributes/methods that are associated with them.
- We can access these attributes by using a period:

  `[var_name].[attribute]`

- There are mutable and immutable types
  - Fancy way to say that you can change objects by accessing

Strings

- Strings contain characters
- Example of some useful string attributes:
  - split – splits a string on particular characters
  - upper – converts everything to uppercase
  - find – identifies specific elements in the string

Strings (cont.)

```
#Small example
x = "Hello World!"
x.split(" ")
```

```
"Hello" "World!"
```

Basic Functions

- `len(x)`
  - Returns the length of an object
  - Equivalent to R's `length()` function

Basic Functions

- `len(x)`
  - Returns the length of an object
  - Equivalent to R's `length()` function
- `set(x)`
  - Returns unique items
  - Equivalent to R's `unique()` function
- Assignment operator: = (**Unlike in R: <-**)

Range Function

- `range(x)`:
  - Returns a sequence of numbers of length `x`, starting from 0
  - Will create a range object – to view the contents, you have to write a for loop
  - Example:
    - `range(10)`
      - Actually returns `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
      - **NOT**: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

Range Function

- `range(x)`:
  - Returns a sequence of numbers of length x, starting from 0
  - Will create a range object – to view the contents, you have to write a for loop
  - Example:
    - `range(10)`
      - Actually returns `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
      - **NOT**: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

Note: The range() object is an iterable, which means to see the contents of it, you have to use a for loop. For example, try: `print(range(10))`.

White Spaces

- White spaces matter!
- There are no brackets in Python, so your computer determines the code folding by indentation/spaces

```r
#In R:
for(i in 1:10){
    print(i)
}
var1<-i
print(var1)

#You can also write:
for(i in 1:10){
    print(i)
} var1<-i; print(var1)
```

```python
#In Python:
for i in range(10):
    print(i+1)
var1=i
print var1

for i in range(10):
    print(i+1)
    var1=i
    print(var1)
```

Looping over Strings

```
#Try these on your own
for letter in "hello":
  print(letter)

my_string = "UCLA MAE"
for letter in my_string:
  print(letter)

count = 0
for letter in my_string:
  count = count + 1
print(count)
```

Lists `[...]`

- Similar to R's numeric vectors (`x <- c(1, 2, 3, 4)`)
- We declare lists using brackets: `[1, 2, 3, 4]`
- Example:

```
#Numbers:
num_list = [1,2,3,4]

#Strings
str_list = ["hello", "world"]

#Lists
list_ception = [[1,2,3], [4,5,6], [7,8,9]]
```

Lists (cont.)

- Common methods:
  - `x.append()`
    - Adds stuff to the end of a list
  - `x.insert(i, x2)`
    - i = index at which we insert x2
    - Inserts element at index i
  - `x.pop()`
    - Takes last element from list and remove it to store it elsewhere
    - Like a stack in C++

Lists (cont.)

- A simple example:

```
list1 = [1, 2, 3, 4, 5]
list1.append([6, 7, 8])
print(list1)
```

```
[1, 2, 3, 4, 5, [6, 7, 8]]
```

- To add elements 6, 7, 8, you can use `.extend()`

```
list1.extend([6,7,8])
```

```
[1, 2, 3, 4, 5, [6, 7, 8], 6, 7, 8]
```

Indexing

- In Python, we index starting from zero
- Example:

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

  - To access the first element of `x` (i.e., 1), what do we type?

    ```
    print(x[0])
    ```

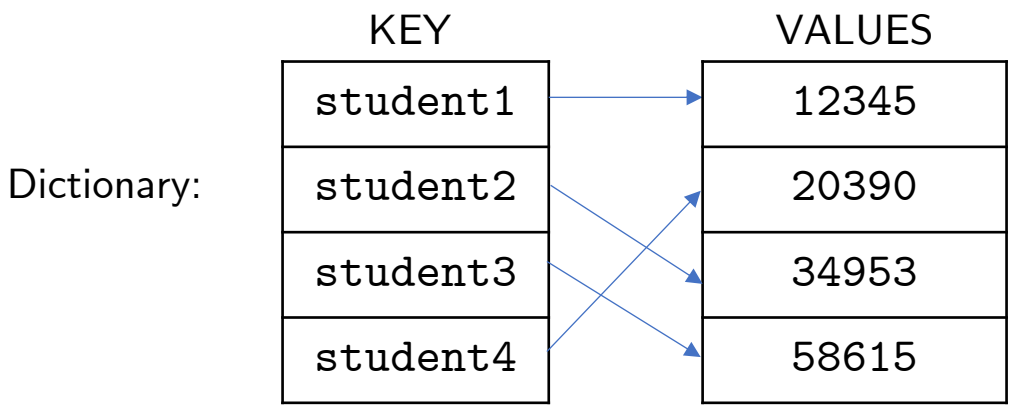
  What if we want numerous elements?

  We can use a colon to access slices of the data.

    ```
    print(x[:5])
    ```

Dictionaries {}

- Python's version of a hashtable
- Each item in your dictionary has both a key and a value associated with it
- Defined by braces: { }
  - More specifically: {key1: value1, key1: value1, ...}



Dictionaries (cont.)

Example:

- Let's say we have a dictionary containing student names and their UID. Call this the registrar.

```
registrar = {'Student1': 12345,
             'Student2': 34953,
             'Student3': 58615,
             'Student4': 20390}
```

Dictionaries (cont.)

Example:

- Let's say we have a dictionary containing student names and their UID. Call this the registrar.

```
registrar = {'Student1': 12345,
             'Student2': 34953,
             'Student3': 58615,
             'Student4': 20390}
```

- So, to retrieve the UID associated with Student 4, we would simply type: `registrar['Student4']`.

Dictionaries (cont.)

- Why use a dictionary?
  - Very computationally efficient to retrieve information!
  - We can input a key, and retrieve the value without iterating through the entire list!

Iterables

- Certain objects in Python are **iterables**, which means that we can automatically loop through them
  - No need to manually iterate through the list with indexing.
- Let `x = [1,3,0,5,9]`. We don't have to call each item of `x` by `x[i]`, we can just say:

```
for i in x:
    print(i)
#Compare the output to this:
for i in range(len(x)):
    print(x[i])
```

Checkpoint

- Print a sequence of numbers from 0 to 100.
- Print a sequence of even numbers from 2 to 100.
- Create a list of even numbers from 2 to 12.

Checkpoint

- Print a sequence of numbers from 0 to 100.

```
for x in range(101):
    print(x)
```

- Print a sequence of even numbers from 2 to 100.

```
for x in range(99):
    print(x+2)
```

- Create a list of even numbers from 2 to 12:

```
numbers = []
for x in range(6):
    numbers.append((x+1)*2)
```

List Comprehension

- List comprehension is a way to define lists and dictionaries in a way where you have a nested for loop within your list.
- Let's say we want to create a list from 1 to 10.
- You could write:

```
Numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Alternatively, using list comprehension:

```
Numbers = [x+1 for x in range(10)]
```

List Comprehension (cont.)

- Provides an elegant way to set up lists
- Let's say you have a really messed up list that contains lists within itself (a list-ception)
- The inner list contains 6 elements:
  1. Open Price
  2. High
  3. Low
  4. Close
  5. Volume Traded
  6. Market Cap

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                …
        [23, 51, 23, 25, 2015, 1034951]]
```

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                …
        [23, 51, 23, 25, 2015, 1034951]]
```

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                …
        [23, 51, 23, 25, 2015, 1034951]]
```

This is the **first inner list** of the larger list.
So to call it, we write: `data[0]`.
But this returns the entire inner list!
We only want the first element of the inner list.

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                …
        [23, 51, 23, 25, 2015, 1034951]]
```

- To call the first element, we treat `data[0]` as if it is any other list, and write: `data[0][0]`.

List Comprehension (cont.)

- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
data = [[42, 53, 10, 25, 2300, 5112346],
        [12, 32, 12, 31, 2600, 1094529],
                …
        [23, 51, 23, 25, 2015, 1034951]]
```

- To access the first elements of the i-th list: `data[i][0]`.

List Comprehension (cont.)

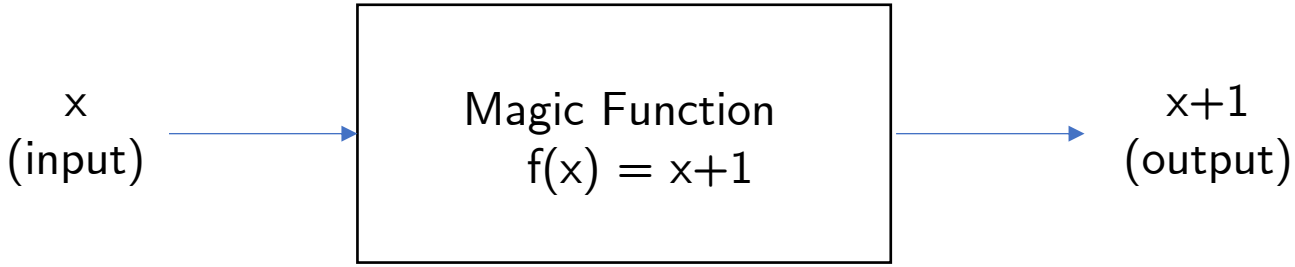
- To separate each element so that you can have just a list that contains all opening prices, you can use list comprehension:

```
#List comprehension:
open = [data[i][0] for i in range(len(data))]

#Equivalent to the following for loop:
open = [] #Empty list
for i in range(len(data)):
    open.append(data[i][0])
```

Functions

- In programming, we use functions so that we can perform repetitive tasks easily.
- Functions take things in as inputs and then produces some sort of result

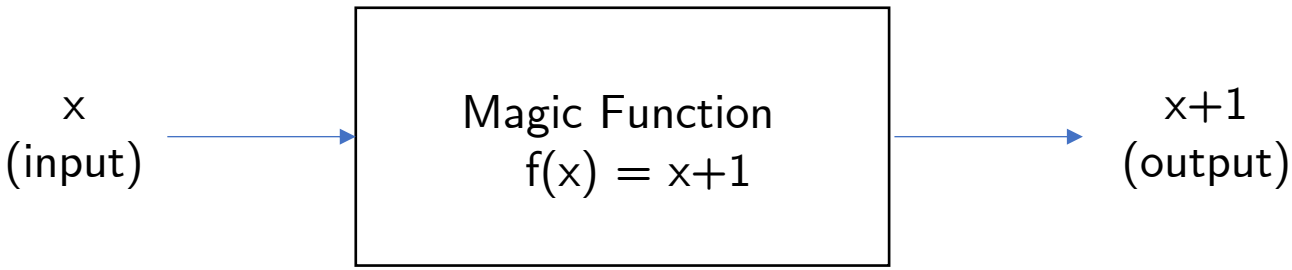


Functions (cont.)

- To declare a function:

```
def functionName(inputVariables):
    [function body]
    return(someVariable)
```

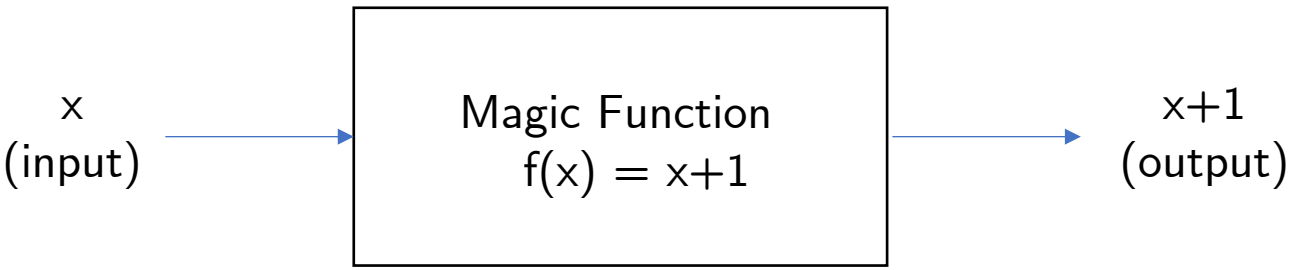
Functions (cont.)



- Programmatically:

```
def magicFunction(x):
    return x+1
```

Functions (cont.)



- Programmatically:

Input variable
(also known as parameters)

```
def magicFunction(x):
    return x+1
```

Indicates to computer that
this is the result (i.e., output)

Functions (cont.)

- When you modify variables within a function, you do not modify them in the global environment.
- As an example, what gets printed out?

```
x = 3

magicFunction(x)

print(x)
```

Functions (cont.)

- So what if we want to store the value that the function outputs?
- Assign the output to a new variable:

```
x = 3

x_new = magicFunction(x)

print(x)

print(x_new)
```