# Python Bootcamp

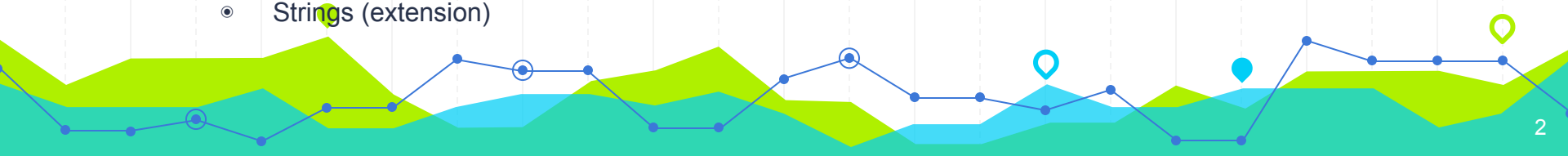
## Workshop 3

8/9

Youth In Code
Justin Liu

# Roadmap

- Functions
- Parameters and Arguments
- Return Statements
- The None value
- Scope
- Lists
- Indexing
- Slicing
- Tuples
- len(), append(), enumerate(), and other miscellaneous list methods
- For loops and lists
- The in and not in Operators
- Augmented Assignment Operators
- Tuples
- Strings (extension)



# What are Functions?

- What do input(), len(), print() all have in common?
- A function allows the grouping of code so that it may be executed multiple times without having to copy-paste the code each time you want to use it
- In other words, functions enable the reusing of code
- Helps deviate from duplicating code
- Makes programs shorter
- Easier to use, read, and reproduce

# Functions (example)

```
❶ def hello(name):
    ❷ print('Hello, ' + name)

hello('Alice')
hello('Bob')
```

Output:
Hello, Alice
Hello, Bob

# Parameters and Arguments

- From the previous slide, the definition of the hello() function in this program has a parameter called name ❶
- Parameters = variables that contain arguments
- When a function is called with arguments passed in, the arguments are stored in the parameters
- Note that arguments are forgotten after the function is done executing
  - Ex. Typing print(name) after hello('Bob') results in an error since name no longer exists. This is due to something known as *scope*
- When hello() was called, it was passed the argument 'Alice' (and ‘Bob’ the second time)
- The program execution enters the function, and since parameter ‘name’ is set to ‘Alice’ (and ‘Bob’ the second time), the console prints ‘Alice’

# Return Statements

- Aside from simply doing things such as printing out text or updating a variable within a function, you can get information by way of *return* statements
- Useful for data retrieval

```
def getSum(num1, num2):
  return num1 + num2

sum = getSum(5, 6)
print(sum)
```

# A quick note on the None value

- In Python, the None value indicates the absence of a value
- Analogous to “null” in Java
- All functions without a return statement secretly return None behind the scenes

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

# More on scope

```
def spam():
❶ eggs = 31337
spam()
print(eggs)
```

```
Console:
Traceback (most recent call last):
  File "C:/test1.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

- eggs var exists only in the *local scope* created when spam() is called ❶
- The local scope is destroyed after the function is done executing, and so the variable eggs is erased from memory
- Variables that are assigned outside all functions and loops are said to exist in the global scope
- Only global variables can be used in the *global scope*
- Local scopes can't use variables in other local scopes
- Global variables can be used in a local scope
- Avoid having global and local vars with the same name

# Even more on scope

- If you ever want to modify the value stored in a global variable from in a function, you must use a global statement on that variable.

```
def spam():
    print(eggs) # ERROR!
❶ eggs = 'spam local'

❷ eggs = 'global'
spam()
```

```
Traceback (most recent call last):
  File "C:/sameNameError.py", line 6, in <module>
    spam()
  File "C:/sameNameError.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

Python sees an assignment statement for eggs in the spam() function ❶ --> considers eggs to be local. But since print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn't exist. Python will not know to use the global variable version. A way around this is to pass in eggs as a parameter.

# Lists

- Most commonly used *data structure* in Python
- A list is a variable that contains multiple values in an ordered sequence
- Lists are similar to ArrayLists in Java, except even more powerful and intuitive
- Contain any type of variable, and however many variables we wish to have
- Mutable

# Lists

- Example of a list: ['cat', 'bat', 'rat', 'elephant'] or ['hello', 3.1415, True, None, 42]
- Lists begin with an opening square bracket and end with a closing square bracket
- Values inside the list are known as items
- Items are comma-delimited/separated

# Indexing

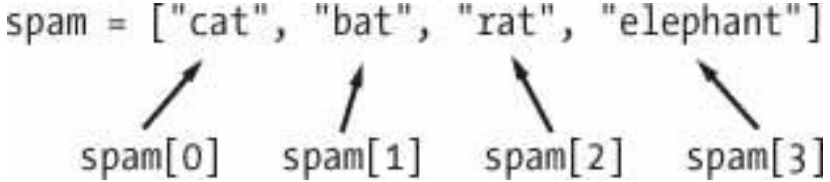
- How do we extract individual items/values within a list?
- An index can be thought of as a tag/id that is specific to the position of a particular item within a list
- Consider the following code:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
```

- What's printed out to the console?

# Indexing

If you guessed *'bat'*, you're correct!




- Indexes begin at 0

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
```

- Make sure not to go out of bounds with your indexes, i.e., your index must be [0, len(list))
- Indexes cannot be floating-point values, only integers

# Nested Lists

- Lists can contain lists as their items

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

# Negative Indexes

- A negative index starts from the end of the list and counts backwards
- Index of -1 corresponds to the last index
- Index of -2 corresponds to the second-to-last index

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-4]
'cat'
```

# Slicing

- A slice returns several values from a list, in the form of a new list
- Typed between square brackets, and it has 2 integers separated by a colon
- Note that the first parameter indicates the starting point of the slice, and likewise for the second parameter (ending point)
- The slice is *not* inclusive of the ending index, but it is of the starting index
- Very rarely you will pass in the 3rd parameter, which denotes the step size

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
```

# More on slices

- You sometimes may choose to leave out one or both of the indexes on either side of the colon in the slice
- Leaving out the first index is equivalent to using 0, or the beginning of the list
- Leaving out the second index is equivalent to slicing to the end of the list

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

```
spam = ['cat', 'bat', 'rat', 'elephant']
>>> print(spam[1:4:2])
['bat', 'elephant']
>>> print(spam[::2])
['cat', 'rat']
print(spam[::-1])
['elephant', 'rat', 'bat', 'cat']
```

# The len() function

- Returns the total number of values/items in a list
- Note that the length of a list will always be 1 greater than its greatest index

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

# Misc.

- You can concatenate lists and replicate them as you do with strings
- Remove values from a list using the del() built-in list function
- The multiple assignment trick (tuple unpacking) lets you assign multiple variables using a list all at once

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
```

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
```

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition = cat
```

# For loops and lists

- To *iterate* through lists, you almost always use a for loop
- Using len(list) as the upper bound for the range() function is convention
- Note that range is non-inclusive of the upper bound and len(list) returns a number that is exactly 1 greater than the last index, making writing for loops very straightforward

```
spam = ['cat', 'bat', 'rat', 'elephant']

for i in range(len(spam)):
  print(spam[i])
```

# The in and not in Operators

- Allow you to check if a value is or is not in a list

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

```
spam = ['cat', 'bat', 'rat', 'elephant']

for i in spam:
  print(i)
```

# enumerate()

- On each iteration of the loop, enumerate() returns two values: 1) the index of the item in the list, and 2) the item in the list
- Useful for getting both the item and the item's index

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

# Augmented Assignments

| Augmented assignment | Regular assignment |
|---|---|
| spam += 1 | spam = spam + 1 |
| spam -= 1 | spam = spam - 1 |
| spam *= 1 | spam = spam * 1 |
| spam /= 1 | spam = spam / 1 |
| spam %= 1 | spam = spam % 1 |

# append()

- In the interest of time, we cannot cover all the list methods out there such as insert(), index(), remove(), sort(), reverse(), etc
- But perhaps the most important function you should know is append()
- append(arg) appends arg to the end of the list
  - Analogous to add() for Java's ArrayList

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

# Tuples

- Same as lists, except they are *immutable*
- All strings are tuples and they can’t be modified
- The aforementioned list methods can all be applied to strings

```
my_string = "string"
my_string[1] = 'j'
print(my_string)

Traceback (most recent call last):
  File "main.py", line 2, in <module>
    my_string[1] = 'j'
TypeError: 'str' object does not support item assignment
```