Python Foundations ivan 2022

1. Programming Basics

Intro

- -Write automated script to solve problems
  -Everything is about Data
  -Must be explicit with our logic
  -Must use language-specific syntax to communicate our logic
  -We must begin by understanding what types of data we have available and how to control the flow of code execution

Pseudocode

- -approach a problem the same way you would provide instructions to a child
  -point is to provide detailed and methodical instructions
  -assume that the child can only keep a few things in mind at a time but can write things down and do some basic math
- -what are your instructions to wash the dished in your home?
  -how would you leave instructions to feed your puppy if you want to feed him/her 1/3 cups of food at 9am, 1/2 cups of food at 1pm, or 3/4 cups of food at 6pm?

2.Data Types

Numbers

- -have the power of MATH
  -5 core math operators:
  + (addition)
  - (subtraction)
  * (multiplication)
  / (division)
  % (modulus) - remainder

  -you cannot mutate the value of a number, you can only reassign

Strings

- str - any character or series of characters
  -we can use either single quotes or double quotes in our definition and usage of strings
  -how t o declare:
  ```
  x = 'some value'
  x = "some value"
  ```

- -strings have the power of concatenation
  -you can combine strings together with the + operator 'hello' + 'corey' #'hellocorey'
  -just like numbers, strings are also non-mutative. That is, you must reassign to change its value

- Booleans
  - bool - for times where there can only be two possible values (occurs often)
    -they can only have the value True or False
    -they posses the power of small space, in some languages booleans only take up 1-bit of data
    -how to declare: x = True

- Helpful Functions
  - check the value of a var that you have been working with in the print function:

    print(YOUR_VARIABLE_HERE)
  - check the type of a var you have been working with the type function.

    type(YOUR_VARIABLE_HERE)
  - Casting
    print (4 / 3) #1
    print (4 / float(3)) #1.333
    print (4 / 3.0)   #1.333

    print ('This is a cool number: ' + str(5))

- Syntactical Tips
  - -use meaningful names for your vars, avoid x = 5
    -vars can be names with any alphanumeric characters,   l33t = 'leet'
    -vars names cannot start with a number
    -multi-name vars should be separated with an underscore (_) ,   puppy_name = 'Kaia'
    -you can leave comments in your code by using the # symbol
    -there is no need for semi-colons to terminate lines in python and their usage is discouraged

3. Control Flow

- Code Execution
  - -an interpreter will read code from top to bottom
    -all code within function blocks is ignored until a call to that function is made
    -sometimes we only want code to execute conditionally (conditional statements)

- Conditional Statements
    - if statements are used to define a block of code that will only execute if the condition is met
      -if that condition fails our execution will skip over that particular block of code
      -indentation is REQUIRED in python to indicate the code related to a particular statment

      ```
      if num > 5:
        print ('Wow num is quite a large number!')
      ```
    - if/else statements
      -else keyword is used to define a block of code that will only execute when the if condition fails
      -if that condition fails execution will run our else block instead of the if block

      ```
      if num > 10:
        print ('Wow num is quite a large number!')
      else:
        print ('num sure is a tiny number!')
      ```
- Logical Comparators
    - equality ==
      not equal !=
      greater than >
      less than <
      greater than or equal to >=
      less than or equal to <=
- Boolean Combinators
    - -and
      ```
        x == 5 and y > 10
      ```
      -or
      ```
        x == 5 or y > 10
      ```
      -not*
      ```
        not (x == 5 or y > 10)
      ```
- Loops
    - for loops

- for loops are used to iterate over a particular, generally, fixed range
  -during the definition of a for loop, we will define a var that will change during each execution of the code block
  -var can represent each character, one by one, in a string
  -var can represent each character, one by one, in a range
  -var can represent each character, one by one, in a list*
  -we will also define a code block that will be un on each execution
  -indentation is REQUIRED in python to indicate any "dependent" code block

-
  ```
  str = 'hello'
  for char in str:
    print (char)  #h e l l o

  for num in range (1, 5):
    print (num)  #1 2 3 4
  ```

- while loops

  - while loops are used to give you fine control over repeated coded execution or indefinite/user-defined code execution

    ```
    x = 10
    while x > 5:
      print (x)  #10 9 8 7 6
    ```

- -we can repeat execution over a finite range or an iterable type using a for loop
  -we can have finer control over repeated execution using a while loop

4. Functions

- Purpose

  - -the recipes of code
    -used for modularity
    -used for expandability
    -single-responsibility principle

- Definition

  - -in python we use the def keyword to define a function block
    -we follow that keyword with the name of the function we are trying to define
    -then we, in parenthesis define what parameters (ingredients) our function (recipe) will need

    ```
    -def some_func(str_1):

        #your function code block
    ```

    - -when defining a function, the func is NOT actually run
      -like a recipe, it is simple a set of instructions ready to run when the ingredients are provided and you are ready to cook
      -we write the func as if we have all the info we need)this is why we define the parameters)
  - Execution
    - -when we execute, invoke, call, or run a function - first time the code within a func block will run
      -at execution we have the opportunity to provide our func the ACTUAL values it will use to complete its task. These are called ARGUMENTS
      -we can execute a func by using the func name, and using parenthesis uncluding the arguments to be passed to the func
    - definition:

      ```
      def adder (num_1, num_2):
          print (num_1 + num_2)
      ```

      execution:

      ```
      adder (4, 5)
      ```
    - Execution context
      -it will pause execution of current work and "bookmark" this position to return to after the completion of the func
      -all vars and data used within a func will, by-default, be unavailable after completion of the func
  - Returning Data
    - -to return a result from a func so that it can be continued to be used elsewhere we will use the return keyword
      -when we return data from a func we also need to remember to capture that data for later use
  - Summary
    - -func definition is like a recipe, we still need the ingredients and go through the actions to make the meal but we can do so at any time
      -during def parameters represent the values a func expects to receive when it will be used in the future
      -execution of a func is the act of suing a particular func and this is where it is our responsibility to provide arguments: the actual values parameters represent
      -if we want to receive a result from a func sent back to the position from which it was called, we can do so with return
- **5. Strings in Depth**
  - A series of characters

- -Each character of a string is assigned an index
  -The indexes of strings always start at 0 and increase by 1, in order, until the end of the string
  -Bracket notation [] is used to access a particular index of a string
   ex: print(string[3]) prints the 4th character in string
  -You can get a substring of a string using slicing:
   Syntax: [start_index(inclusive):end_index(exclusive)]
   Slice to the end of a string like this: [start_index:]
   Slice from the beginning of a string to a specific index like this: [:end_index]
   Slice from the end of a string with negative numbers [-index_from_the_end:]
  - Indexes
    ```
    hi_string = 'hello'
    # index:     01234
    ```
  - Bracket Notation
    ```
    print(hi_string[1])     # prints 'e'
    ```
  - Slicing
    EX 1 - slice from a start_index to an end_index
    ```
    print(hi_string[2:4])   # prints 'll'
    ```
  - Slicing
    EX 2 - slice to the end of a string
    ```
    print(hi_string[1:])    # prints 'ello'
    ```
  - Slicing
    EX 3 - slice from the end of a string
    ```
    print(hi_string[-4:])   # prints 'ello'
    ```
- in python, strings are immutable
  - -Strings cannot be changed (mutated)
    -If you want to change your string, define a variable and use reassignment
    - EX 1 - try to change a string:
      ```
      'antwan' = 'rntwan' # This will throw a SyntaxError
      ```
    - EX 2 - change it the right way:
      ```
      name = 'antwan'

      name = 'p' + name[1:]   # This is how we do it

      print(name)     # prints 'rntwan'
      ```
- strings have methods

- -String Methods always start with a .
  -Built-in methods are lowercase
  -Just like functions, methods must be invoked with parens ()
  -String Methods are called at the end of a string ex: string.upper()
  -String Methods always return a new string, and do not change (mutate) the original string
  - METHOD BASICS

    ```
    food = 'TACOS'

    print(food.lower())     # prints 'tacos'

    print(food)     # prints 'TACOS'
    ```
- .find() METHOD - finds the index of first occurrence of what you are looking for
  - EX 1 - find a char:

    ```
    fav_food = 'tacos'
    # index:    01234

    print(fav_food.find('c'))       # prints 2
    ```

Lists

- What is a list?
  - -non-primitive data structure in python
    -values inside of a list are called elements

  -
    ```
    numbers = [1, 2, 3]
    names = ['George', 'John', 'Thomas']
    a_variable = 'a value'
    mixedBag = [30, True, 'apples', aVariable];
    ```
- What is the type of a list?

  -
    ```
    names = ['George', 'John', 'Thomas']
    print(type(names))
    ```
- Bracket access
  - -access elements in a list the same way you'd access a character string using brackets and the index number corresponds to the position of the element inside of the list

    -
      ```
      names = ['George', 'John', 'Thomas']
      print(names[0])
      print(names[1])
      print(names[2])
      print(names[3])
      ```

- Bracket assignment
    - -lists are mutative so you can modify the value at any given position
      -use brackets and the assignment operator to assign new values to index positions in a list

        -
          ```
          names = ['George', 'John', 'Thomas']
          names[0] = 'Washington'
          names[1] = 'Adams'
          names[2] = 'Jefferson'
          ```
- Getting the length
    - lists, like strings, have a length

        -
          ```
          names = ['George', 'John', 'Thomas']
          print(len(names))
          ```
- Slicing lists
    - slicing is non-mutative and works the same way as it does for a string

        -
          ```
          names = ['George', 'John', 'Thomas']
          oneTermPresidents = names[1:2]
          print(oneTermPresidents)
          print(names)
          ```
- .append method
    - .append adds an element to the end of the list

        -
          ```
          names = ['George', 'John', 'Thomas']
          names.append('James")
          print(names)
          ```
- .pop method
    - .pop removes one element from the end of the list. it returns the removed element

        -
          ```
          names = ['George', 'John', 'Thomas']
          jefferson = names.pop()
          print(names) #notice this is modified
          print(jefferson)
          ```
- .remove method
    - .remove removes the first appearance of the passed in element in a particular list

        -
          ```
          names = ['George', 'John', 'Thomas']
          names.remove('Thomas')
          print(names)
          ```

- .index method
  - .index is a list method that works the same way as the string method .find
    - names = ['George', 'John', 'Thomas']
      print(names.index('George'))
      print(name.index('Alexander'))
- .count method
  - .count takes a value, and returns the number of occurrences of that particular value
    - names = ['George', 'John', 'Thomas']
      print(names.count('George'))
- .reverse method
  - .reverse mutates (changes) the original list, reversing the order of its elements
    - names = ['George', 'John', 'Thomas']
      names.reverse()
      print(names)