Variables

A **variable** is a descriptive name to label and store data in memory. Overall, variables can be thought of as the "data" that we use in our program

Variables can be **assigned** to be either **values** or **expressions** using the `=` operator

- Assigning a variable to a **value**
  - `motto = keviniscool`
  - `x = 69`
- Assigning a variable to an **expression**
  - `product = 420 * 69`
  - `length = len(motto)`

Variables help programmers more clearly understand and read the meaning of code. They also allow for the reuse of common values and in turn, the reusability of code

Practice:

- Create two variables called `length` and `width` with the values `12` and `17` respectively
- Create a variable called `area` that is equal to the area of a 12 x 17 rectangle
- Create a variable called `area` using an **expression** instead of individual **values**

Functions

A **function** is a self contained module of code that accomplishes a specific task by transforming a particular input into a corresponding output. We write functions to generalize a given task so the function can be "called" over and over again in the future

The general form of a function is as follows:

```
def function_name(parameters):
    body
```

- **Function Name** - the name to use in order to "call" the function
- **Body** - the code that the function will execute when called
- **Parameters** - any (optional) input values a function will use

Practice:

- Define a function called `calculate_area` that takes in two parameters `length` and `width` and calculates the area of a rectangle defined by such values
- Define a function called `triangle_area` that takes in two parameters `base` and `height` and calculates the area of a triangle defined by such values

Modules

Most programming languages (Python included) contain many libraries of pre-built functions for your own convenience called **modules**

To get a list of which functions a module contains, in the shell, you can use the command `dir(module)` where `module` is the name of the module you want to check

To find out more about a particular function, in the shell, you can use the command `help(module.function)` where `module` is the name of the module the function resides in and `function` is the name of the function you want to check

To use a module's functions, you must first import the module by writing `import module` at the top of your program replacing `module` with the module you wish to import

Example:

```
>>> python
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', ...
>>> help(math.floor)
...
```

Practice (using the module `math`):

- Calculate `2 * pi`
- Take the floor and ceiling of `56.4`
- Calculate the square root of the logarithm of `244`
- Calculate the base 2 logarithm of `256`

Strings

Python uses a built-in type called `str` which stands for "string". We can iterate and index the characters of a string but the string itself is immutable (we cannot modify the values of a string once created, but we can construct *new* strings)

For the following print statements, recall that the `+` operator concatenates (combines) strings and the `,` operator can be used to separate items in a print statement

String Iteration Practice:

- Assign a string (perhaps `keviniscool`) to the variable `s`
  - Write a loop that prints each character in `s` on a separate line
  - Write a loop that prints each character in `s` on the same line with a space after each character
    - Example - `k, e, v, i, n, i, s, c, o, o, l,`
  - Write a loop that prints each character in `s` on the same line with a comma after each character but without the space after each comma
    - Example - `k,e,v,i,n,i,s,c,o,o,l,`
  - **Bonus**: Redo the last two tasks but without the trailing comma after the last character

A **method** is a special type of function that belongs to a specific data type and is designed to make use of a specific value it is "called on". These functions cannot be called on their own and must be attached to a type in order to be called

For example, the `str` method `upper()` uses the value of the string it is attached to and returns its uppercase equivalent

```
>>> 'keviniscool'.upper()
'KEVINISCOOL'
```

String Methods Practice:

- Check whether or not a string ends with the letter `h`
- Check whether or not a string contains only numbers
- Replace every instance of the character `l` (lowercase "L") in a string with a `1` (the number "one")
- Given a string composed of only numbers, pad it with zeroes so that it is exactly 6 digits long
- Check whether or not a string contains only lowercase letters
- Remove all leading zeroes (0s at the beginning of the string) from a string composed of only numbers

For the following practice, you are **NOT** allowed to use any of Python's `str` methods. However, you may use functions from `__builtins__`

String Functions Practice:

- `longer(str, str)`: given two strings, return the length of the longer string
- `earlier(str, str)`: given two strings made up of lowercase letters, return the string that would appear earlier in the dictionary
- `count_letter(str, str)`: given a string and a single-character string, count all occurrences of the second string in the first and return the count
- `remove_digits(str)`: return a new string that is the same as the given string, but with the digits removed
- `repeat_character(str, str)`: given a string and a single-character string, return a string consisting of the second, single character string repeated as many times as it appears in the first string
    - Example:
        - `repeat_character("keviniscool", "i")` should return `ii` because there are two `i`s in the first string
- `where(str, str)`: given a string and a single-character string, return the index of the first occurrence of the second string in the first; if the second string is not in the first, return -1
    - Examples:
        - `where("abc", "b")` should return 1
        - `where("abc", "z")` should return -1

Lists

Python uses a built-in type called `list` which represents a **list**. A **list** is similar to the concept of "arrays" seen in other programming languages with the differences being that a Python list can contain *any* collection of types (not just a single type) and that when instantiating (creating) a list, you do not need to initially specify the length of the list

We process lists in a similar manner to strings. Suppose we have the list:

```
fruits = ['apple' ,'banana', 'orange', 'cherry', 'grape', 'onion']
```

Then `fruits[1:4]` would return a new list equal to `['banana', 'orange', 'cherry']`

Practice:

- For the following steps, use names and slice notation:
    - Write an expression that produces this new list: `['orange', 'cherry', 'grape']`
    - Write an expression that produces this new list: `['apple']`
    - Write an expression that produces this new list: `['cherry', 'grape', 'onion']`
    - Write an expression that produces a new list containing the only element that is not a fruit
- Given a list `L` and a value `v`, using list methods (i.e. not a loop), write an expression that removes the first occurence of `v` from `L`
- Write an expression that adds the string `"Mr. Stark"` in front of the list `["I don't feel so well"]`
- Write code that transforms `[2, 4, 99, 0, -3.5, 86.9, -101]` into `[99, 86.9, 4, 2, 0, -3.5, -101]`
    - You should use just **two** method calls
    - **Bonus**: see if you can do the transformtion in a single statement
- Write a function `every_third` that takes a list as a parameter and returns a new list that contains every third element of the original list starting at index `0`. Do not use slice notation
- Write a function `every_ith` that takes a list `L` and an integer `i` as parameters and returns a list consisting of every ith element of `L` starting at index `0`. Do not use slice notation

While Loops

A **while** loop loops through an iterable object as long as a given condition is true. Use `while` loops to complete the following functions. Do not use `for` loops. The type of the parameter is given in parentheses. Do not actually use `list` as the name of a parameter or variable

Practice:

- `display_list(list)`: print the elements of a given list
- `display_list_even(list)`: print the elements of the given list that occur at even indicies
- `display_list_reverse(list)`: print the elements of the given list from the end of the list to the front
- `sum_elements(list)`: sum the elements of the given list of ints starting from the front of the list until the total is over 100 or the end of the list is reached and return the same at that point (as an int)

- `duplicates(list)`: return `True` if the given list contains adjacent elements with the same value and return `False` otherwise

Nested Lists

Because a `list` can contain *any* type; it is possible for a `list` to contain an element that is a `list` in itself. When this happens, it is called a **nested list** or a **list of lists**

Example:

```
avengers = [["peter parker", "spiderman", True, 22], ["tony stark", "ironman", False,
53], ["stephen strange", "doctor strange", True, 42], ["steve rogers", "captain
america", False, 37], ["bruce banner", "the hulk", False, 51], ["peter quill",
"starlord", True, 39]]
```

We can access each element of the list `avengers` using its index:

```
>>> avengers[0]
["peter parker", "spiderman", False]
```

We can element each item of the inner lists using the approriate inner index:

```
>>> avengers[0][1]
"spiderman"
```

In the above example, we access the `0th` element of the outer list and then the `1st` element of the inner list to obtain the final value of `"spiderman"`

Practice:

- Write a loop that prints each list from `avengers` on a separate line
- Write a loop that prints the secret identity of each Avenger on a separate line
- Write a loop that examines the list `avengers` and computes the number of Avengers that were snapped by Thanos
    - An Avenger was snapped by Thanos if the 3rd element in the list is `True`
- Write a loop that examines the list `avengers` and computes the sum of their ages
    - An Avenger's age ies the 4th element in the list
- Write a function `nested_lengths` that takes a list `L` as a parameter and returns a list of the lengths of the sublists
    - Formally, for each element `e` in `L`, the returned list contains a corresponding element `c` that represents the number of elements in `e`

Text Files

In this section, you will practice opening, reading, and writing text files.
Use `dir` and `help` to get information on file methods provided by Python

Before you begin, download the text file `data.txt` and the starter code file `file_basics.py`. Implement the three function stubs according to their docstring descriptions

Resources

- Python Docs
- Function Definitions
- More on the OS module