Chapter 6: Data Types, Strings, and Basic Math

Thus far, previous chapters and examples have utilized a number of programming constructs, such as variables, numbers, strings, math, and parameter-passing. However, we have not yet formalized all of these concepts nor explicitly noted related rules and syntax. This chapter will define and discuss these topics, now that we have used them and have some context.

6.1: Data Types

Much like in algebra, a variable in programming is a name that can be used to reference a value stored in memory. For example, the statement `x = 3` assigns a new variable, named *x*, the value of 3 in memory. Similarly, another example might be `person_name="John Smith"`. Here, the variable is `person_name` and the value it references is the value "John Smith".

Until now, we have glossed over the idea that values, and therefore the variables that reference them, have **types**. One such type of data is numeric, and more specifically, **integer** or **float.** An integer is a numeric value with no decimal places. A float is a number that can (but does not have to) have decimal places. Here are a few examples typed directly into the Python (in this case the Spyder) command Console:

```
x = 3

type(x)
Out[38]: int

y = 4.567

type(y)
Out[40]: float

z = "JohnSmith123@somemail.com"

type(z)
Out[42]: str

r="The quote, 'to be or not to be', is famous"

type(r)
Out[52]: str

w=(3,4,5)

type(w)
Out[44]: tuple
```

```
q = [34, "hello", 5.67]

type(q)
Out[46]: list

n={3, 7, 10, 11}

type(n)
Out[48]: set
```

The `type` function will return the type of any data type or data structure. Given the above set of examples, the x variable is an integer type (called int), the y variable is a numerical type called float. The float type is numbers with decimal places. The z variable is a string. A string is always contained in quotes (either single or double). If there is the need to use quotes inside of a string, single quotes can be used on the inside if double quotes are used on the outside. This can also be reversed. Above, the variable r is a string that contains quotes. The variable w is called a tuple, which is a data structure in Python. Tuples are contains in parentheses. The variable q is a list data structure type, and the variable n is a set data structure type. Details about data structures will be discussed in chapter 7. For now, it is important to note that Python offers many different data types and structures. It is also an option to import modules, such as numpy, to gain further data structure options, such as numpy arrays. Numpy will be discussed in chapter 9.

Python cannot automatically convert data from one type to another. However, there are functions in Python that can make some type conversions. For example, the `int()` function will convert an appropriate string to a numerical integer.

```
a = "34"

type(a)
Out[56]: str

x = int(a)

type(x)
Out[58]: int
```

However, Python will return a syntax error and will stop execution if the `int()` function is misused. Here, the string `"bob"` cannot be converted to an integer numerical value.

```
a="bob"
```

```
type(a)
Out[60]: str

x=int(a)
Traceback (most recent call last):

  File "<ipython-input-61-87d65aebf98c>", line 1, in
<module>
    x=int(a)

ValueError: invalid literal for int() with base 10: 'bob'
```

It is also feasible to convert a float (decimal value) to an integer using the `int()` function. When the decimal number is converted to an integer, it is truncated (not rounded).

```
x = 4.67

type(x)
Out[63]: float

y = int(x)

type(y)
Out[65]: int

y
Out[66]: 4
```

Data types must be considered and consistent for function definitions and calls. Recall that a function can have zero or more parameters. Once the function is defined, the types of the parameters delivered to the function via a function call must be identical to the types of the parameters presumed in the function definition. The following example will illustrate this concept.

Example 6.1.1: Incongruent data types in function definition and call

```
## This Program will generate an ERROR

def main():
    sname=input("Please enter your name: ")
    sect=input("Please enter your section number: ")
    Student(sname, sect)
```

```
def Student(studentname, sectionnum):
    if sectionnum <= 2:
        print("Hello ", studentname, "Report to Room 1 for orientation.")
    else:
        print("Hello ", studentname, "Report to Room 2 for orientation.")

main()
```

Here is an input/output for the above program and the error it will create:

```
Please enter your name: John

Please enter your section number: 2

Traceback (most recent call last):

File "<ipython-input-68-6fdde9847892>", line 1, in <module>
runfile('C:/Users/Ami/Documents/Python Scripts/OpenCoder.py',
wdir='C:/Users/Ami/Documents/Python Scripts')

File "C:\Users\Ami\Anaconda3\lib\site-
packages\spyderlib\widgets\externalshell\sitecustomize.py", line 699, in
runfile

execfile(filename, namespace)

File "C:\Users\Ami\Anaconda3\lib\site-
packages\spyderlib\widgets\externalshell\sitecustomize.py", line 88, in
execfile

exec(compile(open(filename, 'rb').read(), filename, 'exec'), namespace)

File "C:/Users/Ami/Documents/Python Scripts/OpenCoder.py", line 22, in
<module>

main()

File "C:/Users/Ami/Documents/Python Scripts/OpenCoder.py", line 14, in main

Student(sname, sect)

File "C:/Users/Ami/Documents/Python Scripts/OpenCoder.py", line 17, in
Student
if sectionnum <= 2:

TypeError: unorderable types: str() <= int()
```

Why did this program generate errors and fail to properly execute?

A hint toward the answer is the last line of the collections of errors that resulted from running the program:

```
TypeError: unorderable types: str() <= int()
```

This states that it is not syntactically correct to compare a string (str) to an integer (int).

In the function definition for `Student`, the first two lines of code are:

```
def Student(studentname, sectionnum):
    if sectionnum <= 2:
```

The function `Student` is defined to accept two parameter values, `studentname` and `sectionnum`. Next, the parameter `sectionnum` is compared to the number 2 (see the statement in bold above).

The comparison requires (and assumes) that `sectionnum` is a numerical type so that it can be compared to the number 2. However, when the program collects the section number inside of `main`, the variable `sect` is collected using `input` (without `eval`). Therefore, `sectionnum` is collected as a **string** (not a number).

```
sect=input("Please enter your section number: ")
```

This program can be corrected by changing this line of code to:

```
sect=eval(input("Please enter your section number: "))
```

The corrected program is:

```
## Corrected Program Option 1

def main():
    sname=input("Please enter your name: ")
    sect=eval(input("Please enter your section number: "))
    Student(sname, sect)

def Student(studentname, sectionnum):
    if sectionnum <= 2:
        print("Hello ", studentname, "Report to Room 1 for
orientation.")
    else:
        print("Hello ", studentname, "Report to Room 2 for
orientation.")
```

```
main()
```

The output is:

```
Please enter your name: John

Please enter your section number: 2
Hello  John Report to Room 1 for orientation.
```

This example illustrates the requirement that data types must match and attention must be paid to them during programming.

There is a second alternative for correcting this program. The following code illustrates another option.

```
## Corrected Program Option 2

def main():
    sname=input("Please enter your name: ")
    sect=input("Please enter your section number: ")
    Student(sname, sect)

def Student(studentname, sectionnum):
    sectionnum=int(sectionnum)
    if sectionnum <= 2:
        print("Hello ", studentname, "Report to Room 1 for
orientation.")
    else:
        print("Hello ", studentname, "Report to Room 2 for
orientation.")

main()
```

The output to this alternative program is:

```
Please enter your name: John

Please enter your section number: 2
Hello  John Report to Room 1 for orientation.
```

In this case, rather than collecting the section number as an integer using `eval`, the function `Student` converts the string version of `sectionnum` into an integer using the `int()`

function. Either method is fine. The key is that the data type of the `sectionnum` variable must be correct.

In a very generalized sense, numbers can be compared to other numbers using logical operators. Similarly, strings can be compared to other strings using logical operators. However, numbers and strings cannot be compared to each other. The end of this section will illustrate a few examples.

```
a = 5

b = 3.678

c = "Fred"

d = "Freddy"

a > b
Out[89]: True

b > a
Out[90]: False

c == d
Out[91]: False

d > c
Out[92]: True

a > c
Traceback (most recent call last):

  File "<ipython-input-93-feb456b97575>", line 1, in
<module>
    a > c

TypeError: unorderable types: int() > str()
```

When the comparison between a (an integer) and c (a string) is attempted, an error is produced.

6.2: Strings and String manipulation

Now is a great time to take a much closer look at strings and string manipulation. String manipulation can be very powerful, especially for areas, such as data mining, data science, and analytics.

6.2.1: String type and changing types

A **string** is a collection of characters, including special characters and numeric characters. Strings must be contained in a pair of matching single or double quotes (either is fine). If a string must contain a quote, this can be done by using double quotes on the outside and single quotes on the inner quote (or vice versa). The following are five examples are all examples of strings. Type each into the Spyder console and check its type using.

Example 6.2.1: Possible string examples.

```
name= "Bob"

ID='HD34_672'

Code="!@44##ABCD"

FullName="John W. Smith"

Shake="The quote is 'to be or not to be' "

SomeSentence="This is a sentence. The entire thing is a
string type!!"
```

To check the type of any variable, use the `type()` function. For example, `type(ID)`, from the variable ID in the above example, results in `str`, for string.

A string can be thought of as a list of characters, symbols, spaces, or numbers. Strings can be compared and combined with other strings, but not with numerical types.

To better understand and review this concept, follow this exercise.

Exercise 6.2.1: Changing Types

In the Spyder Console, follow these steps.

1) Create a variable called, `Var2`, and set it equal to the string value of 34.6

```
Var2="34.6"
```

2) Add 55.7 to Var2 and note this this creates an error and will not execute.

```
Var2 + 55.7
```

3) Next, change the type of Var2 to? in a float (decimal number) and set the result equal to FVar2

```
FVar2=float(Var2)
```

4) Check the type of FVar2. The result should be float.

```
type(FVar2)
```

5) Now, FVar2 can be added to any other number. Confirm this.

6) Change FVar2 from a float to a string

```
FVar2 = str(FVar2)
```

7) Check the type of FVar2. It will now return str for string.

```
type(FVar2)
```

This exercise illustrates that the types of variables can be changed. The int(), float(), and str() functions will return a variable of that type.

8) At this point, `FVar2` is a string. Next, type in the following:

```
float(FVar2)
type(FVar2)
```

What type is FVar2? It is still a string. Why? Because when the `float()` function was applied to `FVar2`, no variable was set equal to the result.

9) Now, try this and confirm that now `FVar2` is a float.

```
FVar2 = float(FVar2)
type(FVar2)
```

10) As the last step, review these four statements and type them into a Spyder console to confirm.

```
num1="34"

num2="5.5"

result=int(num1)+float(num2)
```

```
result
Out[127]: 39.5

type(num1)
Out[128]: str
```

Here, `num1` and `num2` are both strings. But, to get the result, `num1` is **cast** to an integer number and `num2` is cast to a float number. The values are then summed and the answer is placed into the variable `result`. This does not change the types of `num1` or `num2`. Check their types to confirm that they are still strings.

6.2.2: Built-in string manipulation methods

There are many built-in Python functions that can be used to work with strings.

String Length

```
len(<string>)
```

The function `len()` will return the length of any string.

Example:

```
MyString="This is a string!"

stringlen=len(MyString)

stringlen
Out[131]: 17
```

There are 17 characters in the string, `MyString`. The spaces count as a characters, as do the two exclamation marks.

Input

```
<VariableName>=input(<string>)
```

Examples:

The **input** function, which has been used throughout this book, allows the user to input a string.

```
GetInput=input("Enter your first and last name:")
Enter your first and last name:Harry Potter
```

```
GetInput
Out[373]: 'Harry Potter'

type(GetInput)
Out[374]: str

len(GetInput)
Out[375]: 13
```

Here, the variable named, `GetInput` is assigned whatever the user enters. In this example, the user entered Harry Potter. The type of `GetInput` is *str*, and the *len* is 12 (which includes the space between Harry and Potter).

eval

```
NumericalVariable=eval(input(<string>))
```

The **eval** function (for evaluate) will convert a string that was input by the user into a number.

Example:

```
Age=eval(input("Enter your age: "))

Enter your age: 27

print(Age)
27

print(Age + 20)
47
```

The `eval` function is required in combination with the `input` function to collect information in numerical form. Because `eval` is used, the variable `Age` is numerical. However, if only the `input` function had been used, then the variable `Age` would be a string, and could not be directly added to the value 20.

capitalize

```
StringName.capitalize()
```

Example:

```
Name="john smith"
```

```
Name.capitalize()
Out[142]: 'John smith'
```

Here, the "J" in john smith is capitalized.

Similarly, there are many other **string methods** that can affect a given string, including **`upper(),replace(a, b), center(num), count(letter),`** and **`find(item).`** Below is an example that illustrates each of these methods. Notice that methods are called using the **dot "." operator**. We will see more of this when we talk about classes and objects.

Example:

```
SomeSentence="This is a sentence."

SomeSentence.upper()
Out[388]: 'THIS IS A SENTENCE.'

SomeSentence.replace("This", "It")
Out[389]: 'It is a sentence.'

SomeSentence.replace("e", "123")
Out[399]: 'This is a s123nt123nc123.'

SomeSentence.center(50)
Out[390]: '               This is a sentence.
'

SomeSentence.count("a")
Out[391]: 1

SomeSentence.count("e")
Out[394]: 3

SomeSentence.find("is")
Out[395]: 2

SomeSentence.find("e")
Out[396]: 11

SomeSentence.find("sentence")
Out[397]: 10
```

Explanation:

Consider each statement in the above example. The variable called `SomeSentence` is assigned the string, *"This is a sentence."*. This string has 19 characters, including the spaces and the period. The first method in the output above is **upper**. When using a **string method**, the dot operator followed by the name of the method is called. Methods are functions that "belong" to a certain **class**. Classes and objects will be reviewed in details in a later chapter. The statement, `SomeSentence.upper()` will change the entire string to uppercase.

The **replace** method will replace a character or set of characters with the included replacement. The statement: `SomeSentence.replace("This", "It")`, will replace the `"This"`, with `"It"` to give an output of `"It is a sentence."`. Similarly, the statement: `SomeSentence.replace("e", "123")`, will replace all *`"e"`* characters in the string with `"123"`, which outputs, `"This is a s123nt123nc123."`.

The **count** method will count up the number of characters (including symbols and spaces) that match the required item. The statement: `SomeSentence.count("a")` will count all the *"a"*'s in the string. In this example, there is only one "a" in the entire string and so the return value will be 1. As a second example, the statement: `SomeSentence.count("e")` returns the number 3 as there are 3 *"e"*'s in the string. In addition to counting individual characters, the count methods can also count up words or other substrings (smaller groups of characters).

Lastly, this example illustrates the **find** method. This method can locate any string, substring, or character. If it finds the desired string, it will return the number of characters that occur before the located string. For example, the statement: `SomeSentence.find("sentence")` will return the number 10 because there are 10 characters (including any spaces or symbols) that occur before the string of interest, *"sentence"*, starts.

Here are two more examples of the *find* method. The statement: *SomeSentence.find("e")* will return the value 11 because the first *"e"* is after the first 11 characters. The statement: *SomeSentence.find("is")* will return a 2. At first, this seems odd, because maybe we expect it to find the word *"is"* that follows the word *"This"*. However, the first occurrence of *"is"* is within the word "This" and there are 2 characters that occur before the first occurrence of *"is"*.

There are many methods and functions that offer string manipulation. The next concept to cover and explore is the idea that a string is a sequence of individual characters. This offers the option of accessing any part or portion of a string.

6.2.3: String Indexing

A string is a collection or list of characters. Any spaces, numbers, or special characters (such as @ or #) are also considered characters in the string.

Indexing can allow access to any single or group of characters in a string. For example, consider the following string:

```
Book="Fah 451#"
```

Table 6.2.1 illustrates the indexing of this string. Indexing can start with 0 if starting with the first character in the string, or -1 if starting with the last character of the string.

F	**a**	**h**		**4**	**5**	**1**	**#**
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

Table 6.2.1: Indexing of the string, `Book="Fah 451#"`

The variable name is `Book`. The value of `Book` is `Fah 451#`. Each character in `Fah 451#` can be accessed using its index. Indexing is done with square brackets, []. Individual characters and substrings can be indexed. The following statements illustrate a few indexing options for `Book`.

```
Book="Fah 451#"

Book[0]
Out[144]: 'F'

Book[1]
Out[145]: 'a'

Book[3]
Out[146]: ' '

Book[4]
Out[147]: '4'

Book[5]
Out[148]: '5'

Book[6]
Out[149]: '1'

Book[7]
Out[150]: '#'

Book[1:4]
Out[151]: 'ah '
```

Negative and Math Indexing

It is also possible in Python to index using negative integers, as well as math expressions. For example, `Book[-1]` is the character '#', and `Book[-8]` is the 'F'. As shown in Table 6.2.1, the index of -1 is the last character in the string. Simple math can be performed to generate an index as well, either positive or negative. For example, `Book[2-8]` is the same as `Book[-6]`, which is `'h'`. Similarly, `Book[3+5-2]` is the same as `Book[6]` which returns '1'. More complicated calculations are also permitted as indices, as long as the final index is an integer. For example, the following statement: `Book[int(3*2/5)]` is the same as `Book[1]` which will return 'a'.

Substring Indexing

Sometimes, it is more valuable to be able to access multiple characters (or substrings) within a larger string. This can also be done through indexing options. The **colon**, ***":"***, operator can be used to access multiple characters within a given string. In general, the colon operator has the format of *YourString[a:b]*, where *a* is the beginning index location and *b* is the ending index location. All characters from *YourString[a]* through *YourString[b-1]* will be included in the result. *YourString[b]* is not included.

Example:

```
Book = 'Fah 451#'

Book[1:5]
Out[156]: 'ah 4'
```

The following statements and their output illustrate some uses of the colon operator for indexing.

```
Book = 'Fah 451#'

Book[:5]
Out[418]: 'Fah 4'

Book[3:]
Out[419]: ' 451#'

Book[2:5]
Out[422]: 'h 4'
```

In the first example above, `Book[:5]`, will include all characters from the beginning of the string through (but not including) the 5th index. In our example, this will be 'Fah 4'. Recall that "F" is at index 0 and the space is at index 3.

Exercise 6.2.3: String Indexing and Decisions

Review, type in, save, and run the following program. Note how the program uses string indexing to avoid user error. Run this program for several input options. As the user, it is possible to enter as many numbers as you wish. The while loop will repeat until the user enters stop. However, because indexing is used, the user can actually end the program by entering s, S, stop, STOP, Stop, etc. In fact, any word that starts with "s". Also note that if the user does not enter any numbers and chooses to stop right away, the program has an option for this contingency. After running this program, make some changes to see what effects those changes have. This program is reviewed by line number below.

```
# AverageByIndex.py
# Ami Gates

def main():

    total=0
    count=0

    print("This program calculates the average.")
    print("Enter as many numbers as you like. ")
    print("Enter STOP when finished.")

    userinput=input("Enter the next number or STOP to end: ")

    while userinput[0] != "S" and userinput[0] != "s":
        count=count+1
        total=total+float(userinput)
        userinput=input("Enter the next number: ")

    if total > 0:
        average=total/count
        print("The average of your ",count, "numbers is ",
average)
    else:
        print("No numbers were entered.")

main()
```

Possible input and output:

```
This program calculates the average.
Enter as many numbers as you like.
Enter STOP when finished.
```

```
Enter the next number or STOP to end: 75

Enter the next number: 85

Enter the next number: 95

Enter the next number: stop
The average of your  3 numbers is  85.0
```

Review of this program by line number:

```
1. # AverageByIndex.py
2. # Ami Gates

3. def main():

4.     total=0
5.     count=0

6.     print("This program calculates the average.")
7.     print("Enter as many numbers as you like. ")
8.     print("Enter STOP when finished.")

9.     userinput=input("Enter the next number or STOP to end: ")

10.    while userinput[0] != "S" and userinput[0] != "s":
11.        count=count+1
12.        total=total+float(userinput)
13.        userinput=input("Enter the next number: ")

14.    if total > 0:
15.        average=total/count
16.        print("The average of your ",count, "numbers is ",
average)
17.    else:
18.        print("No numbers were entered.")

19. main()
```

Review by Line:

Lines 1 and 2:

The first two lines of code are comments. Normally, it would be best to offer an explanation of what the program does, what input is expected, and what output is expected. This is omitted here for brevity.

Line 3:

There is only one function in this program, and it is the main function. Line 3 is where main is defined.

Lines 4 and 5:

Line 4 creates a variable called total and initializes it to 0. The variable total will be used to collect the sum of all numbers entered by the user. It will also be used to calculate the final average.

Line 5 creates a variable called count, which keeps track of the number of values the user enters. This will also be used to calculate the final average.

Lines 6 – 8:

Lines 6 – 8 print out an explanation of what the program does and how to end it. Line 6 specifies that the user must enter the word "STOP" when they are finished entering numbers.

Line 9:

Line 9 collects the next value, or input from the user as a string. This is an important point. There is no eval used here. Whether the user enters the word stop or another number for the final average, the variable "userinput" is of a? type string.

Line 10:

Line 10 is the while loop that will repeat until the user enters stop. However, the while loop allows the user to enter any word that starts with an s or S. The variable, `userinput[0]` will only be the first letter of the word entered. So, if the user enters, STOP, or Stop, or stop, or sally, the program will end. This helps to avoid trying to determine if the user will enter STOP correctly. Specifically, the statement:

```
while userinput[0] != "S" and userinput[0] != "s":
```

will repeat until `userinput[0]` starts with an s.

Line 11:

Line 11 increments the count variable, thereby keeping track of the number of values entered by the user.

Line 12:

Line 12 adds the next user number to the total. However, notice that `float(userinput)` is needed here because the user input is a string when it is entered. If the userinput is not converted to a float number, it cannot be added to the total.

Line 13:

Line 13 retrieves the next input from the user and starts the process again. If the input is a word that starts with "s", the loop will end. Otherwise, count will be incremented, the value entered by the user will be cast to float and added to the total, and the next input will be retried. This while loop will continue until the user enters STOP as instructed (or any word that starts with an "s").

Lines 14 - 18:

Once the while loop ends, the user must have entered STOP (or something similar). The variable called `total` contains the sum of all the numbers the user entered. However, it is possible that the user entered STOP first and never entered any numbers at all. In this case, total would be "0" and `count` would also be "0". To get the average, we must divide `total` by `count`. However, if count is "0", this will cause a "division by 0" error. Therefore, Line 14 checks to see if the total is larger than 0. If it is, then the average can be calculated and output to the user. If it is not, the program prints that no numbers were entered and then ends.

Line 19:

Calls main and starts the main function.

This example exercise illustrates an application of string indexing, as well as the need to presume and manage possible user errors and inputs.

6.3: Additional String Functions and Methods: Concatenation, Split, and Join

6.3.1: The escape symbol in strings

At this point, it is worth making the side note and recalling that Python allows single or double quotes for the definition of strings.

```
MyString="This is a string"

MyString2='This is also a string'
```

There are also cases for which quotes must be inside of strings. For example, the string: *Shakes="He said, "To be or not to be.""* requires two sets of quotes. As it is written, it will result in a syntax error. It is not permitted to put double quotes inside of double quotes, or similarly, single quotes inside of single quotes.

The value of being able to use either single or double quotes, is the option to use one type of quote inside of the other. For example, the statement*: Shakes="He said, 'To be or not to be' "* will work fine. Notice that the single quotes are used on the inside and the double on the outside. The opposite is also fine, and so this statement will also work: Shakes=*'He said, "To be or not to be" '*.

6.3.2: The escape symbol

The other option for having quotes inside of quotes is to use the **escape** symbol, "\". The following statement uses the escape symbol:

```
Shakes="He said, \"To be or not to be\" "
```

The inner double quotes have the escape in front of them to tell the interpreter to treat them as part of the string.

Being able to manage and manipulate strings can allow for considerable programming versatility.

6.3.3: Concatenation of strings using "+"

Concatenation is an operation that allows two or more strings (which can be single characters, numbers, and special characters as well) to be combined together into one string. The plus, **+**, symbol is used for string concatenation.

```
NewString="apple" + "pie"

NewString
Out[5]: 'applepie'
```

Example 6.3.3: Putting strings togther

```
First="I am "
Second="ready to "
Third="go to "

Sentence=First + Second + Third + "the park."
```

```
Sentence
Out[465]: 'I am ready to go to the park.'
```

Here, the three variables, *First*, *Second*, and *Third*, as well as the string, *"the park."*, are concatenated together and assigned to the variable called *Sentence*. When *Sentence* is output, the result is: *'I am ready to go to the park.'*.

6.3.4: Split and Join

The functions **split** and **join** are methods of the **string class**. They are called using the dot operator.

The split Method

The *split* method will split up a string by a given criteria string. The default criteria string is a space. Any string can be used as a splitter.

```
NewList = MyString.split(<string to split with>)
```

Review the following set of example statements for the split method:

```
SomeWords="Hello Bob, how are you today?"

Tokens=SomeWords.split()

Tokens
Out[468]: ['Hello', 'Bob,', 'how', 'are', 'you', 'today?']

Tokens[3]
Out[469]: 'are'

EmpInfo="Sammy,Smith,333-333-3333,Age,38,Degree, PhD"

Items=EmpInfo.split(",")

type(Items)
Out[8]: list

Items[0]
Out[9]: 'Sammy'

Items[3:6]
Out[11]: ['Age', '38', 'Degree']

Items[0]+Items[4]
```

```
Out[12]: 'Sammy38'
```

In this example, the split method is used to split up a string using any splitter (string to split with). The default is a space. If a string is split using a space, the result is a list that contains each element of the string once separated by a space. Above, the string, `SomeWords` contains six space separated elements. The statement, `Tokens=SomeWords.split()` creates a **list** called `Tokens` with is equal to `['Hello', 'Bob,', 'how', 'are', 'you', 'today?']`. Notice that the comma after Bob is with that element because there was no space between Bob and the comma. In addition, the question mark is with the element, `'today?'`.

The split method can be used with any string type splitter. The string, `EmpInfo="Sammy,Smith,333-333-3333,Age,38,Degree, PhD"` can be split up using a comma. The statement, `Items=EmpInfo.split(",")` creates a list called `Items` that contains the seven elements in `EmpInfo` separated by a comma.

While the next chapter covers lists in detail, lists and strings can both be indexed. Therefore, the statement, `Items[0]` is the first element in the list call `Items`. In addition, the statement, `Items[3:6]` offers the 4th, 5th , and 6th elements in `Items`, which are, `['Age', '38', 'Degree']`.

String concatenation can also be used to choose elements of the original string and put them together:

```
Items[0]+Items[4]
Out[12]: 'Sammy38'
```

The join Method

The **join** method of the string class will join together any collection of strings using any **joiner** string. The join method is also accessed using the dot operator. In addition, the join method should not be confused with the concatenation (+) operator.

```
string=joiner.join(<data struncture>)
```

The best way to understand how the join method works is to review several examples. Review the following statements:

```
names=["Bob", "Sam", "Pat"]

NewString=" ".join(names)
```

```
NewString
Out[25]: 'Bob Sam Pat'

NewString2=",".join(names)

NewString2
Out[27]: 'Bob,Sam,Pat'

NewString3=" AND ".join(names)

NewString3
Out[29]: 'Bob AND Sam AND Pat'
```

In the above set of statements, the variable `names` is assigned a **list** of the names Bob, Sam, and Pat. Consider these three statements:

```
names=["Bob", "Sam", "Pat"]

" ".join(names)

Out[487]: 'Bob Sam Pat'
```

In this case, the space between the two double quotes is the string that is being used to join all of the values in the `names` list together. In other words, the space is the **joiner string**. The result is a string.

This may seem very odd at first, but remember that a space is a string, and the join function is a string method. This means that any string (using the dot operator) can call the method `join()`. It is also important to note that the method `join()` will require exactly one parameter - the data structure to be joined together.

Therefore, the statement `" ".join(names)` uses a space to join together all the elements in the data structure list called `names`. The result is the string: `'Bob Sam Pat'`.

The following statements use the comma and the joiner string. All names in the `names` list are joined together with a comma.

```
NewString2=",".join(names)

NewString2
Out[27]: 'Bob,Sam,Pat'
```

In the last set of statements, the `" AND "` string is the joiner. Notice that this joiner string has two spaces and the AND. The result is that all names in the `names` list are joined into one string with *space AND space* between each.

```
NewString3=" AND ".join(names)
NewString3
Out[29]: 'Bob AND Sam AND Pat'
```

Strings are very versatile and there are many functions and methods available in Python that can be used to manipulate string types.

6.4: Basic Math and Statistics in Python

Python is an incredibly versatile language and there are many Packages for science, statistics, and mathematics. To name only a few, there is NumPy (Package), SciPy (Library – see SiPy.org), Matplotlib, pandas, SymPy, and Statsmodels (http://statsmodels.sourceforge.net/).

This section will focus only on the basic math operations and functions that are available in the standard installation of any Python 3. In following chapters, numpy and matplotlib will be discussed.

Tables 6.4.1, 6.4.2, and 6.4.3 below offer a non-exhaustive list of math operations, math methods, and statistical methods.

6.4.1: Mathematical Methods and the math Module

The first step to utilize math methods in Python 3 is to execute this import statement:

```
import math
```

This statement uses the import command in Python, which allows programs to bring in further functionality in the form of modules, packages, and libraries. Once the math module is imported, either within a program or on the console, access to all standard Python math functions is gained.

Console Example:

```
import math

nFact=math.factorial(5)

nFact
```

```
Out[32]: 120

cosVal=math.cos(math.pi)

cosVal

Out[36]: -1.0
```

The math module offers many built-in methods for math and trigonometry.

The math module can also be imported into a program so that throughout the program math methods can be utilized.

Example 6.4.1: Math module program

```
# MathinPython.py
# Example 1 by Ami Gates

import math

def main():
    getVal=eval(input("Enter a number that you need the
factorial for: "))

    output=math.factorial(getVal)
    print(getVal, " factorial is: ", output)

    GetPower=eval(input("Enter a number to raise to the 5th
power: "))
    output2 =GetPower**5
    print(GetPower, " raised to the 5th power is: ",
output2)

main()
```

Here is a possible input/output for this program:

```
Enter a number that you need the factorial for: 5
5  factorial is:  120

Enter a number to raise to the 5th power: 3
3  raised to the 5th power is:  243
```

The statement in the program called, `import math`, allows the option to include the math module. Once the math module is imported, all of its functions and methods may be used. The program example above uses the `math.factorial` method to calculate the factorial of any number input by the user. The above program also uses the math operator, `**`, to calculate the 5^{th} power of a user input value. Table 6.4.1 shows a list of the most common math operators. Table 6.4.2 shows a list of many of the math module functions and methods.

Table 6.4.1: Math operators in Python:

Operator	Operation	Operator	Operation
-	Subtraction	abs(x)	absolute value
/	division of floats	**	exponent
*	product	%	modulus
+	Addition	//	integer division

Table 6.4.2: Math Methods from the math Package:

math.ceil(x)	Returns the smallest integer that is greater than or equal to x
math.fabs(x)	Returns the absolute value of x
math.factorial(x)	Returns the factorial
math.floor(x)	Returns the largest integer less than or equal to x
math.fmod(x.y)	check x % y as well
math.gcd(x,y)	Returns the greatest common divisor
math.isnan	Returns True if the value is NaN (not a number)
math.isinf(x)	Returns True if x is positive or negative infinity

math.exp(x)	Returns e**x where the ** means "to the power of" and "e" is the natural log.
math.log2(x)	Returns the base-2 log of x
math.log10(x)	Returns the base-10 log of x
math.sqrt(x)	Returns the square root of x
math.acos(x), math.asin(x), math.atan(x)	Returns the arc cosine, sine, and tangent in radians respectively.
math.cos(x), math.sin(x), math.tan(x)	Returns the cosine, sine, and tangent in radians respectively.
math.pi	Returns the value of pi
math.e	Returns the value of the natural log, 2.718281 to available precision.

6.4.2: Statistical methods and the statistics Module

The Python standard library contains several methods and functions for basic statistics. For more advanced statistical analysis, there are several additional packages and libraries available, including Statsmodels (http://statsmodels.sourceforge.net/), which is a Python module that can be used to perform descriptive and inferential statistics, and pandas (http://pandas.pydata.org/), which is a data analysis library.

The standard Python module for statistics can be accessed much like the math module. First, the **`import statistics`** statement should be included at the top of the program to import the statistics module. Table 6.4.3 displays some of the available statistics methods and functions. These tables are not exhaustive, and further information about all available statistics functions and methods can be located on this site: https://docs.python.org/3/library/statistics.html.

Table 6.4.3: Statistics Methods for Measures of Center and Variation

mean()	Calculates the average
median()	Calculates the middle value
mode()	Calculates the most frequent value for discrete data

pvariance()	Calculates the population variance
variance()	Calculates the sample variance
stdev()	Calculates the population standard deviation
pstdev()	Calculates the sample standard deviation

Example: Statistics module in a program

```
# StatsInPython.py

import statistics

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

print("The mean: ", statistics.mean(mylist))
print("The median: ", statistics.median(mylist))
print("The population variance: ", statistics.pvariance(mylist))
print("The population std dev: ",statistics.pstdev(mylist))
```

The output for the above sequence of code is:

```
The mean:  5.5
The median:  5.5
The population variance:  8.25
The population std dev:  2.87
```

6.5: Case Study

In this last section, a Case Study will be created and reviewed. This Case Study will include most of the topics covered thus far and will offer an opportunity to investigate program construction. To maximize the value of the Case Study, first consider how you might program a solution to this question. What might the input be? What might the output be? Consider the structure and flow of the program. Think about using decision structures to enact basic error checking for user input. Build a flowchart on paper and try to write and run the code. Build slowly and run the code often so that errors do not become obfuscated. Then, review one possible solution to the question below. Compare the solution offered below to your solution.

Case Study Description:

Sarah and Juan are thinking of buying a home. Like many first time home buyers, they are a bit overwhelmed with many questions, such as what they can afford and how the purchase will affect their overall finances. In making their decision, Sarah and Juan would like to consider several down-payment, mortgage interest rate, and duration options. They would like to be able to note and consider their larger monthly liabilities, such as their car payment, food bill, entertainment bill (including phone and data plans), and general bills (like utilities and water). They would also like to account for the cost of home insurance, property taxes, and maintenance fees. Ideally,

Sarah and Juan would like to have the option to input their personal data, liabilities and expenses, and related information, and then review several options for purchases (including monthly payments for different down-payment percentages, mortgage durations, and interest rates).

Write a program that would offer this functionality and information. Consider methods of user input, goals of output, and allowing user-choice (as not all users will want to share the same information).

Notes: There are many options for writing this program. Before you review a possible solution below, complete this case study project on your own. As a first step, sketch out your ideas on paper. Think about input and output. Think about the overall goals. Think about the functions, decisions, and flow of the program. Remember, start small and test as you go. Also note that the formula for mortgage payments is in a previous chapter example.

A Solution to the Case Study:

The following code (on the next few pages) represents a possible solution to this Case Study. It is by no means the only solution, or even the best solution. However, it does utilize most of the concepts that have been covered in this book thus far. It also takes a closer look at lists, which are part of the next chapter.

```
# HomePurchaseCaseStudy.py
# Chapter 6
# Ami Gates
# This program collects information from the User and offers output pertaining
# to home purchase/cost options

def main():
    #Explain the program to the User
    print("Welcome to Home-Buyers. ")
    print("This program will collect information about your goals and expenses.")
    print("Next, this program will ask you a few questions and will use your answers
to generate home purchase options.")

    def GetUserInput():
        #Get User information about the number of liabilities
        print("This program offers the option of entering your monthly income, ")
        print("as well as your monthly expenses. However, this is not required.")
```

```
        #initialize include_asset_liab as "No"
        include_asset_liab="No"
        include_asset_liab = input("Type 'Yes' if you would like to first enter these
items: ")
        e_sum=0
        if include_asset_liab[0] == "y" or include_asset_liab[0] == "Y":
            (e_list, e_sum) = GetUserAssetLiab()

        # Get User information about purchase price
        num_homes=eval(input("How many homes would you like to enter prices for? :"))
        house_list=[]
        for h in range(num_homes):
            nexthouse=eval(input("Enter the home price: "))
            house_list.append(nexthouse)
            #print("You entered ", len(house_list), " house prices.")

        #Get information about tax rates, fees, and insurance
        getfees=input("Would you like to enter other monthly fees, such as ins or
condo fees? (Y or N): ")
        if getfees[0]=="Y" or getfees[0]=="y":
            (tax,allfeelist,fsum)=GetAllFees()
            #print(tax, feesum, allfeelist)

        for cost in house_list:
            monthlycost=Mortgage(cost,e_sum,fsum)
            #print(monthlycost)

   # This calls the above defined function
    GetUserInput()
# END OF MAIN


#-----------GetUserAssetLiab Function ---------------
def GetUserAssetLiab():
    #expense_list is an empty list that will be filled with expenses
    expense_list=[]
    expense_sum=0
    m_income=eval(input("Please enter your monthly income: "))
    num_liab=eval(input("Please enter the number of expenses you have each month that
you wish to include here: "))
    for i in range(num_liab):
        #This loop will run as many times as the user has expenses
        next_item=eval(input("Please enter the first expense as a dollar amount:" ))
        #the append method allows you to include another expense into the list of
expenses
        expense_list.append(next_item)
        expense_sum=expense_sum+next_item
    print("The sum of your expenses is: ", expense_sum)
    print("Your income minus expenses for each month is: ", m_income-expense_sum)

    return(expense_list,expense_sum)

#-----------------------GetAllFees Function ---------------
def GetAllFees():
    taxrate=.01
    all_fees=[]
    fee_sum=0

    print("Most states charge a yearly real estate tax. For example, DC is about .85%
and FL is 2%.")
```

```
    print("For simplicity and to offer a fair estimate, this program will use 1% for
property tax.")

    num_other_fees=eval(input("How many monthly fees would you like to enter, such as
condo fees, insurance, etc. ? "))

    for f in range(num_other_fees):
        nextfee=eval(input("Enter a monthly fee: "))
        #print(nextfee)
        fee_sum = fee_sum + nextfee
        all_fees.append(nextfee)
    #print("You entered the following monthly fees: ", all_fees)
    return(taxrate,all_fees,fee_sum)

#--------Mortgage Function ---------
def Mortgage(hc,esum=0,fsum=0):
    # hc is house cost
    #This triple nested for loop calculates monthly cost for 4 difference rates,
    # four different durations (years), and 3 downpayment (dp) options
    #print(esum, fsum)
    for yearrate in [.03, .04]:
        for years in [15, 30]:
            for dp in [.10, .25]:
                mortcost = hc - hc*dp
                mrate = yearrate/12
                months = years*12

                monthlypayment = mortcost*((mrate*((1+mrate)**months))/(((1 +
mrate)**months) - 1))
                yearpercent=yearrate*100

                print("The monthly payment for an interest rate of ",
round(yearpercent,2), "%")
                print("and for ", years, "years, and with a downpayment of
",round(dp*100,2)," %")
                print("will be :", round(monthlypayment,2))

                if esum > 0 or fsum > 0:
                    total = round(round(monthlypayment,2) + esum + fsum +
.01*round(monthlypayment,2),2)
                    print("With the additional fees that you noted as well as 1%
property tax, ")
                    print("the updated monthly total will be: ", total, "\n" )
    return round(monthlypayment,2)

# ------------CALL MAIN ----------
main()
```

The following is a possible input/output for the above program:

```
Welcome to Home-Buyers.
This program will collect information about your goals and expenses.
Next, this program will ask you a few questions and will use your answers to
generate home purchase options.
This program offers the option of entering your monthly income,
as well as your monthly expenses. However, this is not required.

Type 'Yes' if you would like to first enter these items: Yes
```

```
Please enter your monthly income: 5000

Please enter the number of expenses you have each month that you wish to
include here: 2

Please enter the first expense as a dollar amount:300

Please enter the first expense as a dollar amount:400
The sum of your expenses is:  700
Your income minus expenses for each month is:  4300

How many homes would you like to enter prices for? :1

Enter the home price: 400000

Would you like to enter other monthly fees, such as ins or condo fees? (Y or
N): yes
Most states charge a yearly real estate tax. For example, DC is about .85% and
FL is 2%.
For simplicity and to offer a fair estimate, this program will use 1% for
property tax.

How many monthly fees would you like to enter, such as condo fees, insurance,
etc. ? 1

Enter a monthly fee: 600
The monthly payment for an interest rate of  3.0 %
and for  15 years, and with a downpayment of  10.0  %
will be : 2486.09
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  3810.95

The monthly payment for an interest rate of  3.0 %
and for  15 years, and with a downpayment of  25.0  %
will be : 2071.74
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  3392.46

The monthly payment for an interest rate of  3.0 %
and for  30 years, and with a downpayment of  10.0  %
will be : 1517.77
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  2832.95

The monthly payment for an interest rate of  3.0 %
and for  30 years, and with a downpayment of  25.0  %
will be : 1264.81
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  2577.46

The monthly payment for an interest rate of  4.0 %
and for  15 years, and with a downpayment of  10.0  %
will be : 2662.88
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  3989.51

The monthly payment for an interest rate of  4.0 %
and for  15 years, and with a downpayment of  25.0  %
will be : 2219.06
```

```
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  3541.25

The monthly payment for an interest rate of  4.0 %
and for  30 years, and with a downpayment of  10.0  %
will be : 1718.7
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  3035.89

The monthly payment for an interest rate of  4.0 %
and for  30 years, and with a downpayment of  25.0  %
will be : 1432.25
With the additional fees that you noted as well as 1% property tax,
the updated monthly total will be:  2746.57
```

Chapter Summary and Notes

Chapter 6 covered details and specifics for strings, as well as options for math and statistics in Python. The last section contained a Case Study that should be viewed and completed as an exercise to practice all concepts learned and covered thus far in this text.