PYTHON INSTITUTE

PCAP: Programming Essentials in Python

Statement of Achievement

Presented to:

Dan Rimniceanu

Name

During the Cisco Networking Academy® self-paced course, the student has studied the following skills:

- the universal concepts of computer programming (i.e. variables, flow control, data structures, algorithms, conditional execution, loops, functions, etc.)
- developer tools and the runtime environment;
- the syntax and semantics of the Python language;
- the fundamentals of object-oriented programming and the way they are adopted in Python;
- the means by which to resolve typical implementation problems;
- the writing of Python programs using standard language infrastructure;
- fundamental programming techniques, best practices, customs and vocabulary, including the most common library functions in Python 3.

Maciej Wichary
VP, OpenEDG Python Institute

Apr 21, 2019

Date

By completing the course, the student is now ready to attempt the qualification PCAP – Certified Associate in Python Programming certification, from the OpenEDG Python Institute.

www.netacad.com | www.pythoninstitute.org

This course is an introduction to fundamental programming concepts by way of the Python 3 programming language. Python 3 is a high-level interpreted language that has many benefits, including easy-to-read and easy-to-write syntax and powerful libraries that provide additional functionality. Even though Python 3 is a great programming language for beginners, it is also used extensively for practical applications in engineering and data science. This course is intended for people with no or very little prior programming experience. It covers a range of topics, such as data types, control flow, functions, file operations, and object-oriented programming. When you finish this course, you will be able to create Python programs for a variety of applications.

Unit 1: Introduction to Python 3

This unit will introduce you to the Python 3 programming language and cover how to use a platform-independent web-based programming environment to begin writing basic Python scripts. We will introduce basic Python data types, the assignment operator, and how to output data to the screen.

Completing this unit should take you approximately 3 hours.

- Upon successful completion of this unit, you will be able to:
  - use an integrated development environment (IDE) to write simple programs;
  - explain the int and float data types;
  - explain variable assignment;
  - apply basic Python output using the print instruction; and
  - explain the string data type.

- 1.1: Introduction to Python 3
  - Introducing PythonPage

    Python is an ideal language for learning to program. Watch this video to see just how useful Python can be. After this, we will introduce an integrated development environment (IDE) and begin programming. The IDE will make it simple to quickly visualize and experience the results of a set of Python programming instructions.

- 1.2: Accessing the Repl.it IDE
  - The Repl.it IDEURL

    In this course, we will use the Repl.it IDE. This web-based IDE requires no installation, which means that you can begin programming in Python

immediately. We will use this IDE throughout the course. On the page, you should see three windows arranged side-by-side in columns. In this unit, we will focus on the rightmost window. This window has a command line prompt, which is indicated by the `>`. We can type basic Python commands using this command prompt. As we progress through the course, we will start to use the middle and left windows.

- 1.3: Python 3 Data Types: int and float
  - Compare and Contrast int vs. floatPage

    Programming languages must distinguish between different types of data. Since we want to introduce simple Python computations, it makes sense to discuss numerical data. The goal of this section is to introduce two fundamental numerical data types: int and float. The int data type refers to integer data. Numbers not containing digits to the right of the decimal point such as `-10`, `0`, and `357` are integers. The float data type refers to floating-point data. Numbers containing digits to the right of the decimal point such as `1.3`, `-3.14`, and `300.345567` are floating-point numbers.

    After you complete this section, you should be able to explain the difference between int and float. There are deeper reasons (beyond the scope of this course) why you might distinguish between these data types; those have to do with how they are represented within a computer system. Our goal for this section is simply to be able to visually identify integer and floating-point numbers. On this page, you will see more examples of int and float.

    The int data type refers to integer numerical data. In the rightmost window of the Repl.it IDE, use your mouse to click to the right of the > command prompt. Next, type the integer `2` and press the "enter" key on your keyboard. You should see the IDE echo the value back to the screen. Try this a few times with other examples to be sure you understand the int data type.

    The float data type refers to floating-point numerical data. In the rightmost window of the Repl.it IDE, use your mouse to click to the right of the > command prompt. Next, type the floating-point number `2.57` and press the "enter" key on your keyboard. You should see the IDE echo the value back to the screen. Try this a few times with other examples that have been provided to be sure you understand the float data type.

    It is of the utmost importance that you understand the value `2` is of type int, while `2.0` is of type float. On paper, this might not seem like a big difference, but the decimal point is how a computer tells the difference between these two data types.

  - Scientific Notation for Floating Point NumbersPage

    Another way of typing floating-point data is to use scientific notation. For instance, the number 0.0012 can also be entered as 1.2e-3. In this example,

"e-3" means "10 to raised to the minus three power" and this value is multiplied by 1.2; hence, 1.2e-3 is equivalent to 0.0012. Try typing `0.0012` into the Repl.it command line, then try entering `1.2e-3`. You should see that Python views both entries as the same value. Scientific notation is very useful when the exponents are very positive or very negative. In data science, exponents in numbers such as -5.63e127 or 2.134589e-63 would not be uncommon. Rather than writing out numbers of this kind in their full form, scientific notation offers a more compact, readable form for presenting float data. If you need to review scientific notation, watch this video.

- 1.4: Variable Assignment
  - Variables and Assignment StatementsPage

    Computers must be able to remember and store data. This can be accomplished by creating a variable to house a given value. The assignment operator = is used to associate a variable name with a given value. For example, type the command:

    ```
    a=3.45
    ```

    in the command line window. This command assigns the value 3.45 to the variable named `a`. Next, type the command:

    ```
    a
    ```

    in the command window and hit the enter key. You should see the value contained in the variable `a` echoed to the screen. This variable will remember the value 3.45 until it is assigned a different value. To see this, type these two commands:

    ```
    a=7.32
    ```

    and then

    ```
    a
    ```

    You should see the new value contained in the variable `a` echoed to the screen. The new value has "overwritten" the old value. We must be careful since once an old value has been overwritten, it is no longer remembered. The new value is now what is being remembered.

    Although we will not discuss arithmetic operations in detail until the next unit, you can at least be equipped with the syntax for basic operations: + (addition), - (subtraction), * (multiplication), / (division)

    For example, entering these command sequentially into the command line window:

    ```
    a=7.32
    b=a+5
    b
    ```

    would result in 12.32 being echoed to the screen (just as you would expect from a calculator). The syntax for multiplication works similarly. For example:

```
a=7
b=a*5
b
```

would result in 35 being echoed to the screen because the variable `b` has been assigned the value `a*5` where, at the time of execution, the variable `a` contains a value of 7.

After you read, you should be able to execute simple assignment commands using integer and float values in the command window of the Repl.it IDE. Try typing some more of the examples from this web page to convince yourself that a variable has been assigned a specific value.

- Reserved Words and Variable Naming ConventionsPage

  Python has its own set of reserved words that, in general, should not be chosen as variable names. As we dive deeper into the course, it will become clearer how to apply these reserved words. For now, just be aware that your variable name choices should avoid the words on this list. Otherwise, though, you can choose any variable name you like. It is important to think about how variable names should be chosen in practical applications. To help others understand your work, you should choose variable names that fit their applications. For example:

  ```
  account_balance=2034.12
  ```

  might reflect the balance contained in a bank account. You will build the skill of sensibly choosing variable names naturally as you work through more programming examples.

1.5: Basic Python Output Using the print Function

- Using Variables in PythonPage

  The "print" function is a Python instruction that will output variable values and results to the computer screen. This instruction will allow us to automatically output data results to the screen when running a program. This section reviews what we've covered so far and introduces examples of the print function. Try executing these instructions in the command line window to make sure you understand how to use the print function:

  ```
  z=1.45
  print(z)
  ```

  Notice how the print function outputs the value of the variable `z` to the screen.

  It is possible to use the print command in a slightly more sophisticated way. Type this set of commands in the command window:

  ```
  temperature=40
  print('Temperature outside = ', temperature)
  ```

Observe, it is possible to "dress up" the screen output by adding some extra descriptive text. We will see more examples of this later on as we introduce and develop expertise with the string data type. But, as a preview of using the print function with text, try typing this command:

```
print('Hello world!')
```

Congratulations! You have now executed what is probably the most-used example in just about every introductory programming course.

Read this page to see more examples of using the print function. Try typing some of those examples in the Repl.it IDE to be sure you are comfortable with the print function. Consider executing these instructions one after the other (that is, sequentially):

```
var1=22
var1=-35
var1=308
```

What value will the variable `var1` contain after these instructions are executed? You can check your answer by using the print function.

- 1.6: More Python 3 Data Types: str
  - The Basics of StringsPage

    Our introduction would not be complete without discussing how to work with text and character data. In this section, you will learn about the string data type and assigning a variable of type "str". String data is text or character data. To assign a variable of type str, the text string must be placed in quotes:

    ```
    string_example='This is the text string'
    ```

    To see that this variable assignment has taken place, type the command:

    ```
    print(string_example)
    ```

    and you should see the text string echoed to the screen. To assign a variable of type str, you can use either single or double quotes, but single quotes are often the convention of choice. As we dive deeper into the course, many interesting operations and computations using the string data type will be presented. Read this page to learn more about the string variable type. Use the Repl.it IDE to practice some of the string commands.
  - The `type` Function

    The `type` function is an instruction that can tell us the type of a variable after it has been assigned a value. We can practice using this instruction and the print function as we study data types in this section. Try typing these instructions in the command line window:

    ```
    a=5
    print(a)
    print(type(a))
    ```

Try typing this set of commands to see how Python distinguishes between the three data types introduced so far. Notice how the type command is used with the print instruction to tell us the data type after a given variable has been assigned.

```
a=3.4
print('a=',a)
print(type(a))
b=3
print('b=',b)
print(type(b))
c='name'
print('c=',c)
print(type(c))
```

In the Repl.it script/run window, as the number of instructions begins to increase, it can be difficult to type them one by one in the command line. The remedy for this issue is to run several commands at once in what is known as a Python "script". The middle window in Repl.it makes this possible. Try copying the above set of commands and pasting them into the middle window beginning at line 1. If you have done this correctly, you should see words that the Python language recognizes (such as print and true) in blue. Now press the "run" button at the top of the page. You should see the expected screen output appear in the rightmost command window.

Predict the output of the type function for each of these variable assignments:

1. `var1 = 3.214e-12`
2. `var2 = 'Wednesday'`
3. `var4 = -32`

You can check your answers using the print function with the type function as discussed above.

Consider executing these commands in the script window (the middle window in Repl.it starting at line 1):

```
a=3
print('a=',a)
print(type(a))
b=3.0
print('b=',b)
print(type(b))
```

Before pressing the "run" button, predict the output to the screen. In other words, mentally execute this code before running it to make sure you understand the data type concepts presented in this unit.

Now, try executing these commands in the script window (the middle window in Repl.it starting at line 1):

```
a=-3.527e3
print('a=',a)
```

```
print(type(a))
b=-3257.0
print('b=',b)
print(type(b))
```

Again, before you press the "run" button, predict the output to the screen by mentally executing this code to make sure you understand the data type concepts presented in this unit.

```
a=3
print(a)
print(type(a))
b=str(a)
print(b)
print(type(b))
```

In this example, the variable `a` has been *cast* as a string using the "str" command. Sometimes it may be necessary to change the type of a variable. Pay close attention to how the variable type changes in this example:

```
a='75'
print(a)
print(type(a))
b=int(a)
print(b)
print(type(b))
c=float(b)
print(c)
print(type(c))
```

- Values, Types, Variable Names, and KeywordsBook
  Read these examples of using the 'print' and 'type' functions. You can also try to use them in the Repl.it command line.

Unit 2: Operators

This unit introduces Python operators. Using the variable types introduced in Unit 1, this unit will allow us to begin computing using arithmetic, relational, and logical operators. We will also introduce operator precedence and discuss what happens when several operators are applied within a single instruction

Completing this unit should take you approximately 4 hours.

- Upon successful completion of this unit, you will be able to:
  - explain Python arithmetic operators, relational operators, logical operations, and the bool data type;
  - explain operator precedence; and
  - construct programs that apply variable types, operators, and operator precedence.

- 2.1: Arithmetic Operators: +, -, *, /, **, %, and //

- Our programming journey begins by phrasing basic operations found on a common calculator using the syntax of our programming language, Python. In the Repl.it command line window, type these commands:
  - `3+4`
  - `3-4`
  - `3*4`
  - `3/4`

  By hitting the enter key after entering each command, you can convince yourself that you now know how to use Python to add, subtract, multiply, and divide. The numbers 3 and 4 are referred to as "operands" and the +, -, *, / signs are referred to as "operators". More specifically, they are "arithmetic operators" because they are used for arithmetic. As we progress in this unit, other types of operators will be introduced.

  Next, try typing the commands:

```
2**2
2**3
```

  and you should see that the ** operator raises a number to a power. Recall that exponents can be positive or negative. Try typing the commands:

```
2**-2
2**-3
```

  where the negative exponents compute the reciprocal of the two previous calculations. Exponents can also be fractional. For instance,

```
16**(1/2)
```

  computes the square root of 16 and

```
8**(1/3)
```

  computes the cube root of 8.

  While these operators are useful for operating on numbers, programming applications require their use with variables. Copy and paste these commands into the Repl.it run window beginning at line 1:

```
a=3
print('a = ', a)
print(type(a))
b=4
print('b = ', b)
print(type(b))
c=a+b
print(a,'+',b,' = ',c)
print(type(c))
c=a-b
print(a,'-',b,' = ',c)
print(type(c))
c=a*b
print(a,'x',b,' = ',c)
print(type(c))
c=a/b
```

```
print(a,'/',b,' = ',c)
print(type(c))
c=a**b
print(a,' to the power ',b,' = ',c)
print(type(c))
a=2
print('a = ', a)
b=5
print('b = ', b)
c=a+b
print(a,'+',b,' = ',c)
c=a-b
print(a,'-',b,' = ',c)
c=a*b
print(a,'x',b,' = ',c)
c=a/b
print(a,'/',b,' = ',c)
c=a**b
print(a,' to the power ',b,' = ',c)
```

Press the run button and observe the output in the command line window. Observe how the assignment operator (=) can be used to update, change, and overwrite a value contained within a variable. Also, pay close attention to the data type generated by each calculation. Initially, the variables `a` and `b` are of type int. However, if the calculation results in a value that involves a decimal point, Python converts the result to the date type float. For the example above, `c=a+b` results in a variable of type int, while `c=a/b` overwrites a previous value and updates `c` to a variable of type float.

Lastly, we introduce two more operators, % and //, that often prove useful for working with integers:

% remainder after dividing two integers (for example, a%b)

// integer division that gives an integer result by taking the "floor" of the quotient (for example, a//b)

Clear out the commands in the run window, then copy and paste these commands into the run window beginning at line 1:

```
x=19
y=7
z1=x%y
print('The remainder of ',x,'/',y,' = ', z1)
z2=x//y
print('Integer division of ',x,' and ',y, ' = ', z2)
```

Observe that since 19=2*7+5, the variable `z1` contains the remainder and `z2` contains the floor of 19/7, which is 2.

- Practice With Arithmetic OperatorsBook

Practice these programming examples to internalize these concepts.

- 2.2: Operator Precedence and Using Parentheses

  - PEMDASPage

    Execute these two instructions in the Repl.it command line window:

    ```
    8**(1/3)
    8**1/3
    ```

    You should recognize the first instruction from the previous section. Observe that these instructions yield two totally different answers. The reason is that there is an order in which operators are evaluated. The "order of operations" and "operator precedence" define how arithmetic calculations are to be evaluated. As expected from basic arithmetic, exponentiation takes place before multiplication and division which take place before addition and subtraction. In addition, operations are generally read from left to right.

    As an example, consider the expression `8**1/3` as shown above. In this case, exponentiation taking precedence over division means that `8**1` is computed first to obtain a value of 8 and then that result is divided by 3. Hence, the final computed result is `8/3 = 2.6666...67`. Notice that, when parentheses are added around the exponent to form `8**(1/3)`, the exponent evaluates to `1/3` and the cube root of 8 is computed. The term inside the parentheses has higher precedence and takes place before exponentiating. This should be a refresher about the order arithmetic expressions are evaluated in.

  - Practice With Operator PrecedencePage

    Now, we will consider order precedence when operations are phrased within the Python programming language. Before running this set of instructions in the run window, try to predict the value each variable will contain.

    ```
    a=3
    b=4
    c=1
    d=5
    e=3
    f=a+b-c*d+e/d
    g=a+b-c*(d+e)/d
    h=a+(b-c)*d+e/d
    i=(a+b-c)*d+e/d
    ```

    You may use the print statement to verify your answer.

    It is important to note that // and % are considered division operations and, because of that, have precedence equal to * and /. Try to predict the value of each variable in these commands:

    ```
    v=2
    w=3
    x=4
    y=19
    z=23
    a=v**v//x%x+y%w*z//x
    ```

```
b=v**(v//x)%x+y%(w*z)//x
```

Again, you should use insert some print commands to verify your answer.

A useful guiding principle when writing code is: if the order precedence is not clear to you by looking at the expression, use parentheses to make things obvious, since parentheses always take the highest precedence.

- 2.3: Relational and Logical Operators
  - The bool Data TypePage

The bool data type is necessary for the evaluation of logical data (which is inherently different from the numerical or string data types introduced so far). "bool" stands for Boolean data which can only take on one of two possible values: True or False. You should recall from the previous unit that True and False are reserved words within the Python language. Try executing these commands in the Repl.it run window:

```
a=True
print(a)
print(type(a))
b=False
print(b)
print(type(b))
```

You should see that True and False appear in your program as blue text indicating that they are reserved Python words. Furthermore, the data type should be identified as bool. Let's investigate how this data type can be used.

Relational operators are those that compare values in order to determine their relationship. Here is a shortlist of very useful relational operators:

- == Equals
- != Not Equals
- > Greater Than
- < Less Than
- <= Less Than or Equal To
- >= Greater Than or Equal To

Copy and paste these commands into the Repl.it run window:

```
a=2
b=3
print(a==b)
print(a!=b)
print(a>=b)
print(a>=a)
print(a<=a)
print(a>b)
print(a < b)
```

```
<="" p="" style="box-sizing: border-box;">
```

It should be clear why these are called relational operators. Also, note that the only possible answers when using these operators are either True or False.

Be EXTREMELY careful to NEVER confuse the assignment operator (=) with the relational equals (==) operator. They serve two totally different purposes. Copy and paste and run this set of commands:

```
a=2
b=5
a=b
print(a)
a=2
b=5
print(a==b)
```

The assignment operator assigns the numerical value contained in the variable b to the variable a. The relational operator compares the variables a and b for equality which evaluates to the bool data type False.

Finally, there are logical operators that allow us to form combinations of logical expressions. Watch the video for a description of relational and logical operators. Afterwards, you should understand these definitions for logical operators given two boolean variables x and y:

- **not x**: Logical complement: if x is True, not x is False. If x is False, not x is True
- **x and y**: Can only be True if both x and y are True, otherwise evaluates to False
- **x or y**: Can only be False if both x and y are False, otherwise evaluates to True

These operators will be incredibly important in the next unit. For now, we just need to practice using them in order to get used to the syntax.

- Let's try a few more in order to make sure that you are completely comfortable with the use of relational and logical operators. Copy and paste this set of commands into the Repl.it run window. Try and predict the value of each variable before running the code.

```
a=3
b=9
c=5
d=(a<b) and (a<c)
print(d)
e=(a<b) and (b<c)
print(e)
print(not a<b)
f=(b<c) or (c<a)
print(f)
g=(b<c) or (c>=a)
```

```
print(g)
```

Make sure you fully understand the output generated by these commands. The ability to mix relational and logical operators to form composite Boolean expressions is an incredibly important skill that all programmers must master.

- Practice with Relational and Logical OperatorsBook

Try these exercises for more practice with relational and logical operators.

- 2.4: Operator Precedence Revisited
  - Now that we have introduced arithmetic, relational and logical operators, it is important to understand their precedence when it comes to the order of operations. Here is a summary of the operators introduced so far with precedence, ordered from highest to lowest:

Parentheses
** (exponentiation)
*, /, //, % (multiplication and division)
+, - (addition and subtraction)
Relational: ==, !=, >, < , >=, <=
logical not
logical and
logical or
Practice running this set of commands and make sure you try to predict the variable values before looking at the answers generated.

```
a=3

b=9

c=5

d=(a<b) and (a<c)

print(d)

e= a<b and a<c

print(e)

f= b<c and b<a or a<c

print(f)

g= b<c and (b<a or a<c)

print(g)
```

```
h= not b<c and not b<a

print(h)

i= not(b<c or b<a)

print(i)
```

- Operators and ExpressionsBook

  Read this to learn more about operators and expressions.

Unit 3: Input and Flow Control Statements

If you have mastered the previous units, you now have the ability to put together a series of sequential Python instructions capable of performing some basic calculations. You also have the ability to format those results and output them to the screen. This unit covers "program flow control", which is necessary for programs to make decisions based upon a set of logical conditions. Your knowledge of relational and logical operators will be pivotal for applying a new set of Python commands that will enable you to control your program's flow. We will also introduce the "input" command so that the keyboard can be used to input data into a program.

Completing this unit should take you approximately 4 hours.

- Upon successful completion of this unit, you will be able to:
  - explain the differences between programmer-initialized variables and user input variables;
  - write a program that will take string and numerical data from the keyboard;
  - write conditional statements using logical operators;
  - write *for* loops and *while* loops using logical operators and the range function for flow control; and
  - explain how *break*, *continue*, and *pass* statements are used in loops.

3.1: Reading Data from the Keyboard

- The Input CommandPage

  The input command is necessary when we want to obtain input from a keyboard. So far, variables have been initialized or assigned values within a given Python script, with commands such as:

```
a=3
b=27
c=a+b
```

  It is often the case that a program requires keyboard input from the user to perform its function. This is exactly what the 'input' command is for. This

instruction can output a message, and the program will wait for the user to input a value. Try running this command in the Repl.it run window:

```
city=input('Enter the name of the city where you are located: ')
print(city)
```

You should be aware that the input function returns data of type str (that is, string data). If numerical data is required, an extra step must be taken to perform a type conversion using either the int or float commands. Copy and paste this set of commands into the run window and then run code using an input value of 34 when prompted for user input:

```
temperature_str=input('Enter the temperature: ')
print(temperature_str)
print(type(temperature_str))
print()
temperature_int=int(input('Enter the temperature: '))
print(temperature_int)
print(type(temperature_int))
print()
temperature_float=float(input('Enter the temperature: '))
print(temperature_float)
print(type(temperature_float))
print()
```

The variable names have been chosen to emphasize and distinguish between their data types. The input command offers the convenience of creating user-defined values within a program. At the same time, it is important to make sure the input value's data type matches the intended use of the variable.

- 3.2: Using Conditional Statements
  - Conditional statements are necessary for a program to make decisions based upon a set of logical conditions. There are three main constructs in Python for this: the if statement, if-else statements, and the if-elif statement.

In Repl.it, run this block of code that checks a logical condition based upon the value of an input variable:

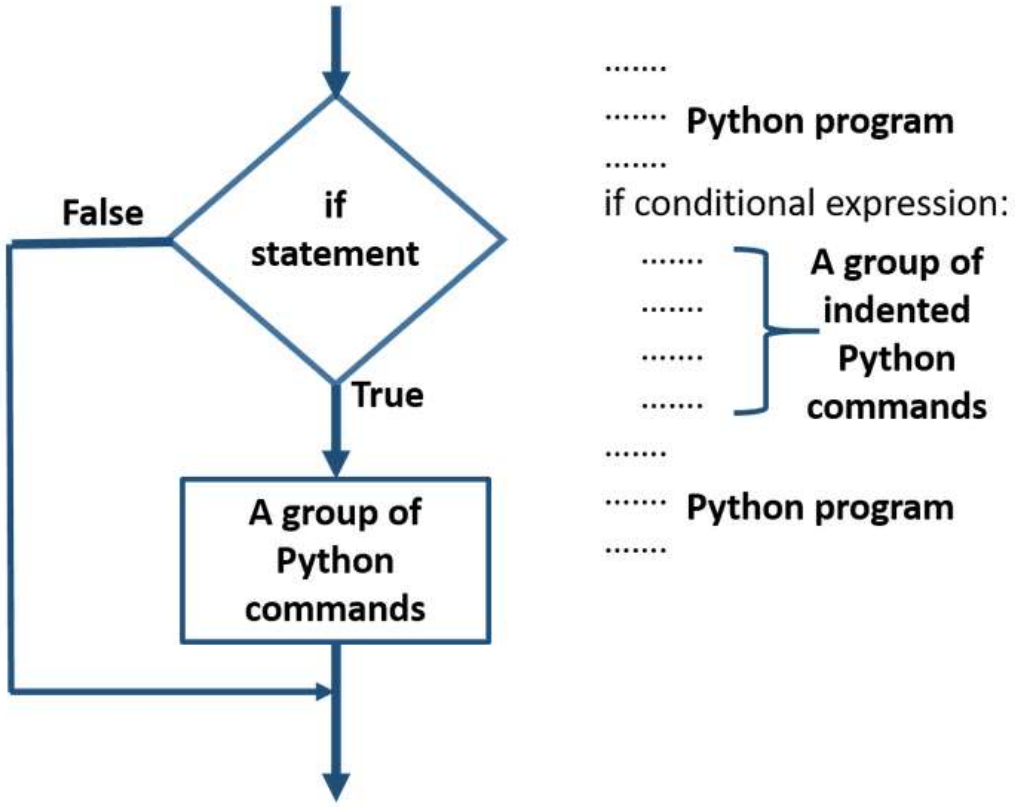
```
avalue= int(input('Enter an integer greater than 10: '))
if avalue>10:
    print('Thank you')
    avalue=avalue/2
    print('This is your value divided by 2: ', avalue)
print('Did your if code execute?')
```

Try running the code first for an input value of 14 and then run it again for an input value of 5. The if statement checks the relational condition '>' to see if the variable `avalue` is greater than 10. If so, the indented code will be executed. If not, the program will not execute the indented code.

Notice the syntax required for the if statement to work:

- The logical expression following the "if" must end with a colon (:)
- The code to be executed if the logical condition is true must be indented. Indentation is how Python knows you have a group of commands inside the if statement
- Indentation can be accomplished in Repl.it using the "tab" key. In the Repl.it editor, this is equivalent to typing two spaces. Some Python editors prefer four spaces for indentation.

This flowchart illustrates how the if statement works:



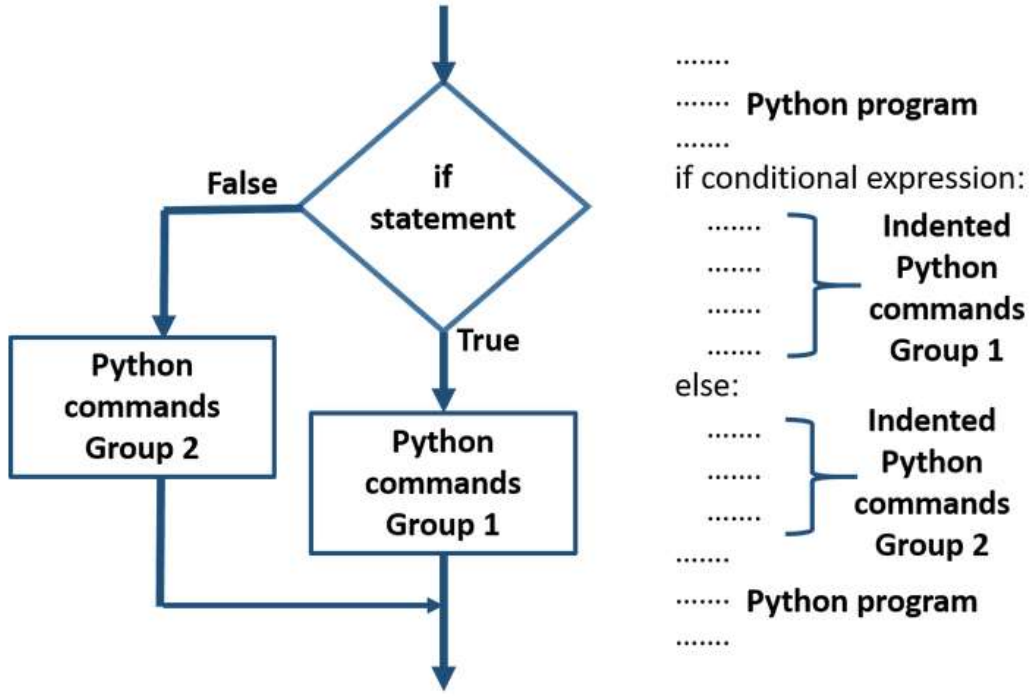
When the if statement is encountered, if the conditional expression is true, then the indented group will be executed. Otherwise, the program will jump over the indented group and not execute those commands. In other words, it is possible to have code in a program that never executes because a logical condition has not been satisfied. It should now be clear why if statements are called "program control" statements: they control the logical flow of the program.

Next, we have the if-else statement, which allows for groups of code to be introduced for both the true and the false condition. Run this code snippet in Repl.it a few times using different values for the input:

```
avalue= int(input('Enter an integer greater than 10: '))
if avalue>10:
```

```
    print('Thank you')
    avalue=avalue/2
    print('This is your value divided by 2: ', avalue)
else:
    avalue=avalue/5
    print('This is your value divided by 5: ', avalue)
    print('Which code group executed?')
```

You should notice different program outputs depending on the logical expression. Again, notice the indentation and notice both the "if" and "else" statements must be terminated with a colon (:). This flowchart illustrates how the if-else statement works:

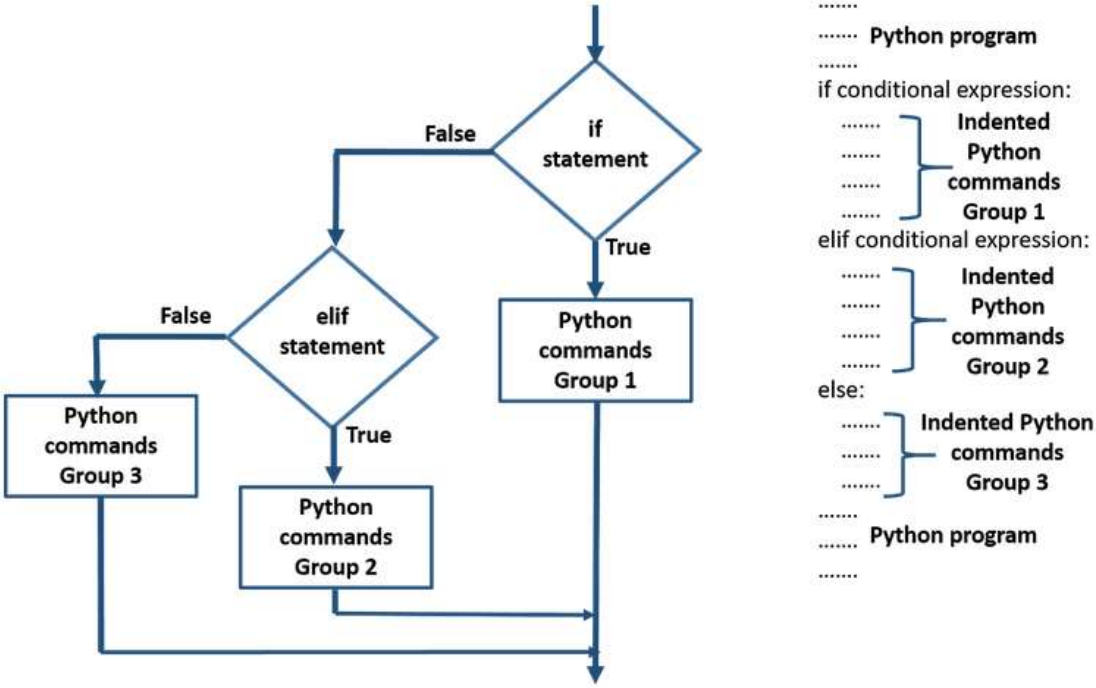


Finally, the if-elif statement allows for several logical conditions to be checked. Run this code snippet in Repl.it a few times using different values for the input:

```
avalue= int(input('Enter an integer greater than 10: '))
if avalue>=10:
    print('Thank you')
    avalue=avalue/2
    print('This is your value divided by 2: ', avalue)
elif (avalue<10) and (avalue>4):
    avalue=avalue/5
    print('This is your value divided by 5: ', avalue)
else:
    avalue=avalue*100
    print('This is your value times 100: ', avalue)
print('Which code group executed?')
```

Notice, once again, the indentation and the colon that are necessary to make the if-elif work execute properly. Also, observe how a series of conditions using logical

and relational operators are checked to control which group of code will execute. This flowchart illustrates how the if-elif statement works:



The if-elif construct allows for several logical conditions to be checked using relational and logical operators. Understanding how to use if, else, and elif statements is central to becoming a seasoned programmer. Make sure to practice the examples you see here in Repl.it.

- if, else, and elif StatementsBook

Read this for more on conditional statements.

- 3.3: Loop and Iterations

  - Using "while" LoopsPage

Loops are necessary when we need to perform a set of operations several times. The first type of loop we will discuss is called the while loop. The while loop checks a logical condition like the if statement. If the condition is true, the code inside the while loop will execute until the condition being checked becomes false. Run this code in Repl.it:

```
avalue= int(input('Please enter the number 10: '))
while (avalue != 10):
    print('Your input value is not equal to 10')
    print('Please try again: ')
    avalue= int(input('Enter the number 10: '))
```

```
print('Thank you')
print('You entered a value of 10')
```

This while loop checks to see if the value input was equal to 10. If not, the while loop will continue to execute the indented code until a value of 10 is input. The syntax rules are similar to those of the if statements:

- The logical expression following the "while" must end with a colon (:)
- The code to be executed if the logical condition is satisfied must be indented.
- **Indentation** is how Python knows you have a group of commands inside the while statement Indentation can be accomplished in Repl.it using the "tab" key. In the Repl.it editor, this is equivalent to typing two spaces. Some Python editors prefer four spaces for indentation.

Make sure you run the examples provided in Repl.it. When we execute the code within a loop over and over again, the process is known as **iteration**.

Using "for" LoopsPage

The for loop is useful when we know how many times a loop is to be executed. Rather than basing the number of iterations on a logical condition, the for loop controls the number of times a loop will iterate by counting or stepping through a series of values. Run this snippet of code in Repl.it:

```
for i in range(5):
    print(i)
```

The `range(5)` command creates a series of five values from 0 to 4 for the variable `i` to cycle through. The statement `for i in range(5)` effectively means use the variable `i` to count through five values starting at 0 and stopping at 4 (Python, by design, begins counting with the value 0). Also, once again, pay close attention to the placement of the colon (:) at the end of the for statement as well as the indentation. The syntax rules should look very familiar at this point. They are exactly the same as for if statements and while loops.

Consider this next snippet of code:

```
n=8
sumval=0;
for i in range(n):
    sumval = sumval+i
    print('Adding ', i,'to the previous sum = ',sumval)
```

This loop counts from 0 to `n-1`, where `n` has been set to a value of 8. An initial sum is set to zero and then successive values of the variable `i` are added to the sum each time an iteration of the loop takes place. In other words, the loop counter variable `i` can be used within the loop as it is being updated. As will we see, the loop variable is often used as part of some computation within the loop. Here is

an example of how powerful Python can be when using a for loop to cycle through a series of values.

```
strval='forever'
e_count=0
for letter in strval:
    if letter =='e':
        e_count = e_count + 1
print('The letter e occurred ', e_count, 'times')
```

Notice how the "if" statement is "nested" (and, therefore, indented) within the "for" loop. The variable `letter` cycles through each character in the string `strval`, which has been initialized to the string `forever`. If an "e" is found while iterating through each character, the count variable is incremented. When the loop is finished, the number of occurrences is output to the screen.

It is also possible to nest loops within loops. We will discuss more of these types of loops as we go deeper into the course. For now, run this code in Repl.it:

```
m=4
n=3
for i in range(m):
    for j in range(n):
        print('i=',i,' j=',j)
```

This is called a nested "for" loop because the inner "for" loop depending on the variable `j` is nested within the outer "for" loop, which depends on the variable `i`. This should be clear from the indentation of the inner loop with respect to the outer loop. For each iteration of the variable `i`, the variable `j` cycles through all of its possible values which depend upon `range(m)`. In order for `i`to be updated to its next value, the inner loop must iterate through all of its values. Make sure you can relate the program output shown the screen to how the nested loop iterates through its values.

After you watch the video and test the examples in Repl.it, revisit the file in the "while" loop section and read through the section on "for" loops. Make sure to practice the for loop examples in Repl.it.

- "break", "continue", and "pass" StatementsPage

Sometimes it is necessary to exit a loop abruptly when a certain condition is met. The "break" statement is sometimes useful for such an application. Consider this code:

```
strval='forever'
for letter in strval:
    if letter =='e':
        break
print('The letter e has been found')
```

In this case, the break statement terminates the loop if the character `e` is found in the `strval` string.

On the other hand, encountering a given condition within a loop may require skipping over a set of commands and effectively "fast-forward" to the next iteration. The "continue" statement is useful for these applications.

Finally, the "pass" statement within a loop is a null statement that does nothing. It is sometimes useful when placed in a program where code is intended but has not yet been written. Read this page and practice the examples.

- 3.4: Further Study
    - More Useful VideosPage

      Here are some supplementary videos that you can review if you'd like a bit more practice the concepts in this unit.

Unit 4: Data Structures I – Lists and Strings

Most of the programming concepts presented so far can be found in any programming language. Constructs such as variable definitions, operators, basic input and output, and control flow via conditional statements and loops are fundamental to what it means to compute. In this unit, we begin studying how data is structured within Python so we can program efficiently. Specifically, you will be introduced to lists and also immersed more deeply in the subject of strings. Upcoming units will introduce even more powerful data structures.

Completing this unit should take you approximately 6 hours.

- Upon successful completion of this unit, you will be able to:
    - explain lists and indexing;
    - write simple programs that apply list and string methods;
    - explain and apply slicing; and
    - write programs that plot and visualize list data.

- 4.1: Python Lists
    - Creating ListsPage

      A list is a data structure capable of storing different types of data. Run this set of commands in Repl.it:

```
a=2
b=3
alist_examp=[1,3.4,'asdf',96, True, 9.6,'zxcv',False, 2>5,a+b]
print(alist_examp)
print(type(alist_examp))
print('This list contains ', len(alist_examp), ' elements')
```

      You should see the list output to the screen as well as the data type of the variable `alist_examp`. Notice that the list contains bool, int, str, and float data.

It also contains the result of relational and arithmetic operations. Any valid data type in Python can be placed within a list.

The data contained within the list are called **elements**. The list you printed contains 10 elements. You can figure this out by counting them by hand. There is also a command called len that will give you this information, which you can see in the example. len is a very useful command.

Pay very close attention to the syntax used to build the list. The left square bracket/right square bracket is known as a "container". Elements must be separated by a comma when building the list.

Pay very close attention to this example:

```
tmp=alist_examp
print(tmp)
print(alist_examp)
tmp[3]='vbnm'
print(tmp)
print(alist_examp)
```

Notice that changing the value in the variable tmp also changed the value in the variable alist_examp. Initially, this appears to go against what we have learned about the assignment operation (=). For reasons that will be discussed in a later unit, when it comes to lists, Python views both variables as occupying the same space in memory. They are different names that refer to the same data. How then can we truly assign the list to a new variable that occupies a different place in memory and can be modified independently of the variable `alist_examp`? This set of commands will accomplish this:

```
tmp=alist_examp.copy()
print(tmp)
print(alist_examp)
tmp[3]='vbnm'
print(tmp)
print(alist_examp)
```

This example introduces some new syntax that will soon become second nature. The command "alist_examp.copy()" is your first introduction to object-oriented syntax. The "copy()" portion of this command is what is known as a **method**. It is connected to the variable alist_exmp using the period or 'dot notation'. This command makes a true hard copy of the variable alist_examp so that modifications of tmp do not affect alist_examp.

Every variable type in Python is what is known as an "object". You will learn an immense amount about objects as we delve deeper into the course. The main point to realize is that there are several methods available for use with lists. The `copy` method we just introduced is one of them. Some other important ones worth mentioning at this point are `pop`, `append`, `remove`, and `insert`. Pay attention to the sections that describe these methods in detail.

- Continuing with our previous example, run this set of commands in Repl.it:

```
print(alist_examp[0])
print(alist_examp[1])
```

```
print(alist_examp[2])
print(alist_examp[3])
print(alist_examp[4])
print(alist_examp[5])
print(alist_examp[6])
print(alist_examp[7])
print(alist_examp[8])
print(alist_examp[9])
```

The elements contained within a list can be referenced using what is known as an **index**. Notice that Python begins indices by starting at a value of zero. So, if a list has 10 elements, the first element on the list is referred to using an index of 0, and the last element is referred to with a value of 9. This can take some getting used to if you are used to counting starting with the number 1.

A common error when first starting with lists is attempting a command such as:

```
print(alist_examp[10])
```

on a list with 10 elements. Such a command would yield an error message and halt program execution because there is no such element.

The index is the key to referring to an element within a list. You must see the programming equivalence between an element and referencing the element via its index. Continuing with our example:

```
print()
c=3 + alist_examp[1]
alist_examp[1]= c +alist_examp[0]
print(c)
print(alist_examp)
```

The whole point of using a list is that a programmer plans to reference elements further down in a program. In this case, 3 is being added to the element with index 1, and then that element is being assigned a new value by referencing the element with index 0. Mastering the gymnastics of using indices is key to becoming an advanced programmer.

In this example, we explicitly typed out all the indices of every element. What if we had a list with 1000 elements, and we wanted to output them all one-by-one? Would we have to type 1000 commands? If we did, we would be completely disregarding the power of loops, which we mastered in the previous unit. Consider this code as an alternative:

```
for i in range(len(alist_examp)):
    print('Element',i, '= ', alist_examp[i])
```

Notice how the result of the `len` command is being used to define the range of the loop. Lists, indexing, and loops are related topics. It is important to understand how they are related to become a seasoned programmer.

One more important point must be mentioned about list indexing. Recall that any valid data type in Python can be inserted into a list. Lists are a valid data type; therefore, it is possible to have a list that contains lists as this example shows:

```
x=[3,4,5.5,6,7.9]
y=[-300,3.14]
z=[x , y, 3.45678]
print()
print(z)
print()
print('The list z contains ', len(z),' elements')
print()
for i in range(len(z)):
    print('Element with index ',i,' = ',z[i])
```

After running this code, you should see that the list z contains 3 elements. The lists x and y are said to be nested within the list z. Therefore, indexing elements within these lists will require a second index as follows:

```
print(z[0][2])
print(z[1][0])
```

where the second index refers to elements within the nested list. To print out all elements on a nested list, we could also use a loop:

```
for j in range(len(x)):
    print(z[0][j])
```

One very useful feature of Python is its ability to reference elements in loops without the need to reference an index, as this example shows:

```
for value in z:
    print(value)
```

In this example, the variable `value` iterates across all the values in the list z without the need to create an actual index. This is a very powerful feature in the Python programming language. Practice as many examples as you can. It is important to master indexing before we move forward to slicing.

- IndexingBook

  Read this for more on indexing.

- SlicingPage

  It is often the case that a program requires referencing a group of elements within a list (instead of a single element using a single value of the index). This can be accomplished by **slicing**. Consider running this example:

  ```
  x=[101,-45,34,-300,8,9,-3,22,5]
  print()
  print(x)
  print(x[0:9])
  ```

  The colon operator can be used to index multiple elements with the list. The index on the left side of the colon (:) is the starting index. By Python's

indexing convention, the value on the right side of the colon (:) indexes that value minus one (be careful with the right index). Since the variable `x` has 9 elements indexed from 0 to 8, `x[0:9]` references all indices from 0 to 8. Here are some more examples you should try:

```
print()
print(x[3:4])
print(x[3:5])
print(x[3:6])
print(x[3:7])
print(x[3:8])
print(x[3:9])
```

Again, you should be careful when using the right index since Python will sequence up to that value minus one. For the sake of convenience, there are also shorthand slicing techniques that do not require specifying the start or ending index when it assumed the complete rest of the list is being referenced:

```
print()
print(x[3:])
print(x[:4])
```

In addition to specifying the start and stop indices, you can also specify the "step". Here is an example that will count by twos from element 3 to element 7 (every other element starting at index 3). The step size is given by the index value after the second colon.

```
print()
print(x[3:8:2])
```

Finally, Python allows for negative indices where, by convention, the index -1 implies starting from the end of the list.

```
print()
print(x[-1])
print(x[-1:-3:-1])
print(x[-1:-len(x)-1:-1])
```

In these examples, the step size of -1 means to count backwards in steps of -1.

This video reviews many of the list concepts we have discussed so far. At 6:11, it discusses and elaborates on the rules for slicing in Python. Be sure to follow along and practice the examples in Repl.it.

- List MethodsPage

Practice these examples using Repl.it to become more familiar with some methods commonly applied to lists. As you go through these examples, you should begin to see how powerful Python can be as a programming language.

- List ComprehensionPage

Python offers many opportunities for creating efficient programming structures. When it comes to writing loops, 'list comprehension' allows for very compact code. It is possible to pack quite a bit of power using just one line using lists. This is an optional topic that requires understanding how to write loops using lists. Practice the examples to expand your ability to write loops.

- 4.2: Strings Revisited
  - List concepts such as methods, indexing, and slicing are also important for dealing with string objects. While there are some similarities in the syntax of processing strings and lists, there is a major difference in how Python views strings versus lists. Lists are mutable, which means that once created, as we have already seen, the elements contained in a list can be changed and updated. Strings are immutable objects. This means that once they are created, they cannot be changed, such as by using assignment operation (=). Because of string immutability, this example would yield an error:

```
a='asdf'
print(a[1])
a[1]='3'
```

While indexing a given character is fine, like by using the `print(a[1])` command, the `a[1]='3'` command will generate an error because it attempts to modify an immutable object.

There are lots of operations we can perform on strings. We will often use the "+" operation, which can be used to join two strings together. When applied to strings, the + sign does not mean add; it should be interpreted as either a "join" or "append" operation. Consider running this example:

```
a='good'
b='morning'
c=a+b
print(c)
print(len(c))
```

As you learn more, you will be able to determine what an operation is based on its context. In this case, when used with strings, you should see that the + operation has joined the two strings together to form a new string.

  - Going Deeper with StringsBook

    Read this for more on strings.

  - String MethodsPage

    There are a host of methods available for processing string objects. Here are a few examples. At this point it is sensible to introduce the comment character, #. The comment character allows you to put comments into your code for the purpose of documentation. While comments are not considered executable

code, they are extremely useful for making your code readable and understandable.

```
#explore changing to uppercase and lowercase
a='good'
c=a.upper()
d=c.lower()
print(c)
print(d)

#join a list of strings together with a space in between the strings
b='morning'
e=' '.join([a,b,'today'])
print(e)

#find a string within a string
#find method returns the first index where string was found
x='a picture is worth a thousand words'
x1=x.find('picture')
print(x1)
x2=x.find('worth')
print(x2)
x3=x.find('words')
print(x3)

#split up a string into a list of smaller strings
#use the ' ' space character as the boundary (delimiter) to split up
the string
y=x.split(' ')
print(y)
print(type(y))

#try the replace method ...
z=x.replace('thousand', 'million')
print(x)
print(z)
```

Take some time to explore your own versions of these examples.

- 4.3: Data Visualization Application
    - The matplotlib LibraryPage

In this section, you will have a chance to exercise your understanding of lists and indexing and apply your knowledge to plotting data. You will also learn how to import libraries into Python. A library in Python generally contains several methods that can be used to perform a specific set of functions. The matplotlib library contains a host of methods useful for plotting data. Try running this snippet of code in Repl.it:

```
import matplotlib.pyplot as plt
x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y=[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
plt.plot(x,y)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Test Plot')
```

```
plt.show()
plt.savefig('plot1.png')
```

The `import` command instructs Python to import the matplotlib library. The extra qualifier `as plt`is added so that the name of the plotting object can be shortened to `plt`. This set of commands plots the data contained in the `x` list against the data contained in the `y` list. The methods `xlabel`, `ylabel`, and `title` are useful for adding annotation to the plot. For completeness, the `show` method is given so that users understand that this command is useful for rendering the plot in many different Python IDEs. However, Repl.it is a web-based IDE, and the `savefig` command will be more appropriate and useful for the rendering of data plots. The leftmost Repl.it window is where you can find the plot file 'plot1.png'. Click on that file to view the plot generated by this code. To return to your Python code, click on 'main.py' in the leftmost window. We will discuss the uploading and downloading of files as we delve into the course more deeply. For now, realize that, for each new plot you would like to generate, you can use the savefig method with a different filename in single quotes (such as plot2.png, plot3.png, and so on).

Make sure to mirror and practice the examples provided in the video tutorial in Repl.it.

Unit 5: Functions

The understanding of variable definitions and control statements is fundamental to any programming language. Furthermore, the syntax of basic data structures such as lists and strings is foundational to mastering the Python language. In this unit, we take another step to improve upon our program organization skills by introducing functions. When a given task is performed many times throughout a program, it is usually wrapped within a function so that it can be used or "called" whenever needed. This notion of creating a specific function or "procedure" to achieve a given task is part of a programming methodology known as "procedural programming". We will also briefly contrast this approach with the use of methods that are used in "object-oriented programming". We will discuss these in greater detail in Unit 10.

Completing this unit should take you approximately 4 hours.

- Upon successful completion of this unit, you will be able to:
  - analyze situations where using functions could improve a program;
  - explain how functions are defined and how to use function syntax;
  - use functions to better organize programs written in Units 2–4;
  - apply functions and methods available from the Python *math* library; and
  - apply functions and methods for generating and applying random numbers.

- 5.1: The Basics of Functions

- Functions are useful when we have a section of code that we need to use over and over again. Putting the section of code in the form of a function allows the reuse of code by calling the function. Functions are provided with input values, they then perform a computation using the input and, finally, return values as the function output. Python has many *built-in* functions such as `print`, `type`, and `len` (which you already have experience with). For example, in the case of the `len` function, the input is a list and the output is the number of elements contained in the list.

Methods are functions that can be used with objects. You have already used a number of methods associated with lists (such as `append`) and strings (such as `find`). We will learn about the creation of methods when we get to object oriented programming. But, it should be pointed out that the idea of using a method is similar to that of a function in the sense that methods have input values and return output values.

The beauty of a programming language is that, while the language has a finite number of commands, keywords and built-in functions, it is possible to create new *user-defined* commands using functions and methods. As an example, assume you have a list of numerical data and you would like to find a value on that list. Assume further that you have a program that must perform this computation at many different points within the program. Under these circumstances it is sensible to write a function to solve this problem. Let's practice writing a function to find a value on a numerical list.

```
#====================
#begin function definitions ...

def findval(alist,x):
    #alist is the input list
    #x is the value being searched for
    #This function returns a bool True if found
    #and returns a bool False if not found
    for val in alist:
        if (val==x):
            return True
    return False

#end function definitions
#====================
#main code begins here ...

a=[2,3,4,5,6,7,8]
print(findval(a,4))
print(findval(a,29))
b=[45,34,78,89]
print(findval(b,45))
print(findval(b,1470))
```

Copy, paste, and run this code in Repl.it to inspect and study its effect. There are many details to be discussed regarding the syntax of functions.

- The keyword `def` informs Python that a function is being declared
- Notice the (:) and the indentation with rules similar to conditional statements and loops
- Function name: The name of the function is `findval`
- Inputs: `findval` has two input variables, `alist` and `x`
- Outputs: `findval` returns one output variable of type bool
- The return keyword is responsible for returning the output value

Now that a new *user-defined function* has been created, the main code that follows can **call** the function as many times as needed and the function will respond to the specific input variables provided. By default, variables used within the function are **local** to the function and cannot be seen by the main commands that follow. The function cannot "see" the variables `a` and `b`. The main code that calls the function cannot "see" the variables `val`, `alist`, and `x` within the function. By convention, functions are defined *at the beginning* of a Python program. The main code that runs comes after all the functions have been defined.

Creating FunctionsBook

Read this for more on functions.

5.2: Some Useful Modules

Trigonometry ReviewURL

Before delving into some Python libraries dealing with applied mathematics, visit this optional site to review trigonometry if you need a refresher.

Random NumbersURL

Programming also involves simulating random experiments such as flipping coins or rolling dice. If you need to review these topics, visit this optional site for a refresher.

The "math" ModulePage

We have already seen using the import command for importing the matplotlib library and the numpy package. Much of Python's power rests in the vast collection of packages, libraries, and modules available for just about any application you could think of. The math module contains a host of mathematically oriented methods typical of what a programmer would need to perform basic calculations. Consider executing this set of instructions:

```
import math
    a=math.exp(1)
    print(a)
    b=math.pi
    print(b)
```

```
x=100
print(math.log(x,10))
print(math.log10(x))
y=math.pi/2
print(math.cos(y))
print(math.sin(y))
y=8
z=1/3
print(math.pow(y,z))
```

Notice once again the object-oriented dot notation for calling a method. Just about any function that can be calculated on a sophisticated calculator can be performed using the math module. For the sake of simplifying the code, if you know exactly what methods are needed from a given library, you can select a subset using the `from` keyword. Consider the code from earlier rewritten using the `from` keyword:

```
from math import exp, pi, log, log10, cos, sin, pow
a=exp(1)
print(a)
b=pi
print(b)
x=100
print(log(x,10))
print(log10(x))
y=pi/2
print(cos(y))
print(sin(y))
y=8
z=1/3
print(pow(y,z))
```

Now all the math method calls look exactly like function calls. Sometimes it is more convenient to use the reduced representation.

- The ability to generate pseudo-random numbers is central to programming and simulation. The random module offers a set of methods for doing so. Try running this code:

```
import random as rn

#set the seed to system clock time
rn.seed()

#test some methods in the random module
a=rn.random()              #uniform random number between 0 and 1
print(a)
b=rn.uniform(7,20)#uniform random number between 7 and 20
print(b)
c=rn.randint(100,200) #random integer between 100 and 200
print(c)
```

Observe that:

- For convenience, the as keyword along with the import command shortens the name of the module reference from `random` to `rn`.
- The "seed" of a random number generator defines the starting point. In actual applications, you want to set the seed to a random value (like the system clock time); otherwise, a pseudo-random list will always start with the same value and repeat itself.
- Three essential methods for generating random numbers are:
    - `random.random()` - generate floating-point numbers between 0 and 1
    - `random.uniform(a,b)` - generate floating-point numbers between `a` and `b`
    - `random.randint(a,b)` - generate integers between `a` and `b`

- 5.3: Application
    - Cryptographic ApplicationPage

      While this example is not how we would perform encryption in practice, it is highly instructive for reviewing concepts covered so far regarding user-defined functions. Implement this example in Repl.it in order to solidify your working knowledge of Python.

Unit 6: Basic Data Structures II – Tuples, Sets and Dictionaries

Python's power lies not only in the vast set of modules and libraries available (such as matplotlib, random, math, numpy, etc.), but also in the data structures that are fundamental to the language. This unit introduces three more ways of structuring data that must be mastered: tuples, sets, and dictionaries. We will also revisit the concepts of mutability and immutability, as we saw for lists and strings.

Completing this unit should take you approximately 3 hours.

- Upon successful completion of this unit, you will be able to:
    - explain the difference between lists and tuples;
    - explain sets and apply set operations;
    - create dictionaries using *dict*; and
    - write programs that apply tuples, sets, and dictionaries.

- 6.1: Immutable Collections
    - Tuples and SetsPage

      We have introduced the concepts of mutable and immutable objects in Python. Recall that lists are mutable while strings are immutable. This section introduces two more immutable data types useful for holding collections of objects: tuples and sets.

In short, tuples are immutable lists. Similar to strings, they cannot be changed once they have been assigned. Everything that you have learned so far about indexing and slicing also applies to tuples. The syntax for forming a tuple uses parentheses (instead of square brackets for forming a list):

```
a=[23.7, 45, 78]
b='asdfasdf'
c=(4,9,a,b,'zxcv')
print(c)
print(type(c))
```

Notice that we can put any object we please into the tuple defined as the variable `c`. The output after running this code looks almost exactly like a list. However, because of the immutability of `c`, attempting to modify an element with an instruction such as `c[1]=99`will result in an error. One obvious use for tuples for holding data that must be write-protected. We will see other uses when we encounter dictionaries.

Sets are another immutable data type that holds a unique collection of objects. Sets can be used to perform typical set operations such as union, intersection, set differences, and so on. Duplicates values are not possible within a set. Left and right squirrely brackets are used to assign sets.

```
a={3,4,5,6,7,9}
print(a)
print(type(a))
b={3,3,3,4,4,4,5,5,6,7,8}
print(b)
print(type(b))
```

You should observe that both `b` and `c` contain the same set of objects. The next set of commands should demonstrate Python's ability to perform set operations.

```
a={3,4,5}
b={4,5,6,7,8,9}
c=a.intersection(b)
print(c)
d=a.union(b)
print(d)
e=8
print(e in b)
print(e in a)
f=b.difference(a)
print(f)
```

This code introduces fundamental operations such as set membership (using the `in` keyword) and a suite of set method calls for computing the union, intersection, and so on.

- 6.2: Mutable Collections
  - Dictionaries structure data to allow for more interesting accessions beyond simple indexing. A given dictionary entry consists of both a key and a value. The key is

then used as a means of accessing a value. This code shows how to create a dictionary and how to access values by using the key:

```
adexamp={'NY':'Albany', 'CA': 'Sacramento','MA':'Boston'}
print(adexamp)
print(type(adexamp))
```

The dictionary is of data type `dict`. Each entry is of the form **key:value**. A value in a dictionary is referenced by its key. The syntax for forming the dictionary requires using left and right curly brackets. It should never be confused with a set data type, as the syntax for the entries is completely different.

Dictionary values are mutable; values can be referenced and modified by using the key.

```
adexamp={'NY':'Albany', 'CA': 'Sacramento','MA':'Boston'}
print(adexamp)
print(type(adexamp))

print(adexamp['NY'])              #value is referenced by the key
adexamp['NY'] ='Buffalo'   #values are mutable
print(adexamp['NY'])              #value is referenced by the key
```

Notice how the "indexing" of the value occurs using the key. It is of extreme importance to note that, while values are mutable, keys are immutable. Any attempt to modify a key after a dictionary has been constructed will result in an error.

However, dictionaries are mutable in the sense that we can add or remove entries.

Continuing the example:

```
adexamp['MD']='Annapolis'  #to add a single new entry
print(adexamp)
bdexamp={'AK':'Juneau', 'AZ':'Phoenix'}
adexamp.update(bdexamp)    #add several new entries
print(adexamp)
del adexamp['MA']
print(adexamp)
```

This example shows the syntax for adding a single entry by invoking a new key. To add several entries, it is more efficient to use a dict method called `update`.
The `del` command is one way to remove an entry by referencing a specific key.
This section will give you more insight into using Python dictionaries.

- DictionariesBook

Read this for more on dictionaries.

- Dictionaries and LoopsPage

One major application of dictionaries is to be able to efficiently sequence through a set of values. Read this section to practice writing loops using dictionaries.

- Dictionaries and TuplesBook

Another application of tuples is to use them as keys in dictionaries. Follow and practice the examples presented in this section in order to understand how tuples can be used with dictionaries.

- 6.3: A Comprehensive Review of Data Structures
  - Getting Started with DataBook

We have introduced several basic Python data structures: lists, strings, sets, tuples and dictionaries. Take some time to review, compare and contrast these constructs for handling various kinds of collections.

Unit 7: File Handling

It is all well and good that data can be created within a program via variable assignments and user input. However, we must also be able to deal with data stored in files. In this unit, we will introduce methods for reading data from and writing data to a file. At its heart, Python is an object-oriented language. Pay attention to the syntax we use here, which will prepare you for the rest of the course.

Completing this unit should take you approximately 2 hours.

- Upon successful completion of this unit, you will be able to:
  - use file handling and file handling modes to read and write to text files;
  - write programs that use file handling modes, such as reading from, writing to, appending, and creating files;
  - write programs that using file handling methods; and
  - apply file handling to the data analysis and visualization programs written in Units 3–6.

- 7.1: File Input and Output
  - File input and output (or File I/O) is the ability to read data from and write data to files stored in a location such as a directory or a folder. The ability to handle files is actually a pretty deep subject that requires some measure of interaction with the computer operating system. Fortunately, for high-level languages such as Python, the nuts and bolts of file I/O are absorbed into a relatively simple set of methods. There are three major steps to referencing a file:
    1. Open the file:
       This lets the operating system know the name and location of the file being referenced and how the file is to be used (such as read or write)

2. Perform operations on the file data (such as read, write, or append):
   Now that the operating system has opened the file, it is ready to be used for the purpose specified in step 1
3. Close the file:
   After the desired set of operations has been completed, the operating system must be informed that access to the file is no longer necessary.

Here is an example of the three steps you can try in Repl.it:

```
fhandle = open('examp.txt','w')
fhandle.write('This is a write example. ')
fhandle.write('Text will be sequentially written until a newline
control character occurs. \n')
fhandle.write('Then a new line will begin with \n')
fhandle.write('and another new line, etc \n')
fhandle.close()
```

The syntax for implementing the above step is fairly straightforward:

4. The "open" command creates and opens a file "examp.txt" where "w" means that data will be written to the file. If we were to read data, we would use an "r" instead (the "r" is actually optional where, if omitted, a file read will be assumed). In short, the first argument to the "open" command is the file name, and the second argument indicates the operation to be performed. An object (named "fhandle" in this example) is created that allows access to a host of methods that will be practiced in this unit.
   In Repl.it, the file location will be in the leftmost column under the "main.py" reference. Since Repl.it is web-based, this column effectively acts like your local directory from which files can be downloaded or uploaded.
   You should notice that when the "open" command executes, the file "examp.txt" is created. The file is still empty, but it is now ready to be written to.
5. Since the file was opened with the parameter "w", data will be written to the file. This operation can be accomplished using the "write" method.
6. Finally, the "close" method closes the file. After writing the data to the file, you should be able to click on the filename in the left window and see the text that was written in step 2.

Try adding the following code to the above script:

```
f2 = open('examp.txt','r')
print(f2.read())    #the 'read' method reads the file
f2.close()
```

You now have the ability to create a file, as well as read data from and write data to a file. The files source.txt, source2.txt, and source3.txt are provided here. You should upload them into the leftmost window in Repl.it for your code to reference them for a file read. You can then practice the examples on this page.

- More on File Input and Output
  - source.txt
  - source2.txt
  - source3.txt

  Download folder

- Syntax and UsageBook

  Now that you're familiar with file input and output, read this for more on syntax and usage.

- 7.2: Visualizing Data from a File
  - The next project will require a couple of steps to set up. First, download these files. Then, Start a new Repl.it session and either upload or 'drag and drop' these three files into the leftmost window. The "csv" stands for comma-separated values. This is a common data file type that is readable by programs such as Excel. We do not need to grab the other data files as we will challenge the Repl.it graphics capability with these three files.

    Once you can see the data files listed in the Repl.it leftmost window, feel free to copy the code in the example provided into the Repl.it run window and run the code. This example is very instructive as it ties together the reading of multiple data files and the use of numpy combined with matplotlib introduced earlier in the course.

    After the code runs, you should see a graph appear in the rightmost window. On the graphic, click on the resize box in the upper left-hand corner to resize the figure. Make sure you see plots similar to those given in the example.

    - Visualizing Data from a File
      - inflammation-01.csv
      - inflammation-02.csv
      - inflammation-03.csv
      - python-novice-inflammation-data.zip

      Download folder

  - Data Visualization from a Data FilePage

    After you download the files above, complete this exercise.

Unit 8: Regular Expressions

At this point in the course, you should have some familiarity with applying string methods for finding a pattern within a string. Regular expressions are a syntax framework for performing more general pattern searches that allow for a measure of pattern variability. The subject of regular expressions is actually quite deep and highly relevant to the theory of computation. This unit will introduce you to the re module and its regular expression syntax to gain expertise with string pattern searches.

Completing this unit should take you approximately 3 hours.

- Upon successful completion of this unit, you will be able to:
  - explain why and how regular expressions are used;
  - use regular expressions to construct search patterns to match a string or set of strings; and
  - solve common tasks by using regular expressions to match patterns.

- 8.1: The "re" Module
  - Syntax and UsageBook

    A regular expression (or "regex") is a character sequence used to search for patterns within strings. You've already seen examples of pattern searching when we looked at strings. Regular expressions have their own syntax, which enables more general and flexible constructs to search with.

    The "re" module in Python is the tool that will be used to build regular expressions in this unit. Practice these examples to familiarize yourself with some common methods. You will also practice building more general regular expression patterns using the table of special characters.

  - Delving DeeperBook

    The subject of regular expressions is quite deep, and it takes an immense amount of practice to get used to the special character syntax. Furthermore, the re module contains a vast set of methods available for performing searches using regular expressions. Upon completing the examples in this section, you should have a much deeper appreciation for how powerful regular expressions can be.

- 8.2: Processing File Data
  - Now, we will practice regular expressions and file handling. First, download this text file. After you download this file, upload it into the leftmost window in Repl.it to ensure file reads will be able to access the data.

    -  Processing File Data
      -  melville-moby_dick.txt
  - Download folder

- Processing File DataPage

  Watch these videos and follow along using the file you just downloaded.

Unit 9: Exception Handling

Any programmer should be able to identify the source of potential errors and implement code to handle those errors. This unit introduces the syntax necessary for achieving this goal. Handling errors can be a sensitive topic because the programmer must address points where something could go wrong. Devote yourself to these examples, as they will be important in your journey to becoming a professional programmer.

Completing this unit should take you approximately 4 hours.

- Upon successful completion of this unit, you will be able to:
  - explain how exceptions are implemented capture programming errors; and
  - write programs that handle errors by using *try*, *except*, and *finally* statements.

9.1: Catching and Handling Errors

- Stuff HappensPage

  When writing software, as you may have experienced already, errors can occur. As we gain experience, we will learn to anticipate what kinds of errors can occur during program execution. For example, when performing a division calculation num/den between two variables "num" and "den", it is possible for the variable den to contain a value of zero. If this happens, a "division by zero" error should be generated to avoid a catastrophic calculation. Exceptions are a way of catching and handling potential errors so that a program will not crash and stop executing in an abrupt, untimely fashion.

  The video provided in this subsection is essential for new programmers because the types of errors that can occur must be identified, discussed, and understood. The discussion on handling errors will be used as a springboard for understanding the rest of this unit.

- Example: Try and ExceptPage

  The "try" and "except" commands are the gateway for handling errors. "try" allows you to try to execute your code, and "except" identifies the type of error to be handled. Python contains many built-in exceptions that can be raised using the "except" command. For example,

  - **ValueError**: raised if an operation or function receives an argument that has an inappropriate value
  - **NameError**: raised if a variable name that does not exist is referenced

- Example: ZeroDivisionErrorPage

  This example shows how to handle a division by zero error by raising the built-in **ZeroDivisionError** exception. Make sure to practice executing this code in Repl.it.

- Example: Else and FinallyPage

  Well-structured code attempts to identify everything that can go wrong as well as everything that can go right. The "else" and "finally" commands, while optional, are important components of professionally written code. The "else" command can be used as part of a "try-except" block to execute code if no errors in the set of errors being tested have occurred. The "finally" section of code will always run, but it helps to programmatically delineate such code from a software organization perspective to distinguish it from code that is being checked for errors.

- Exceptions LessonBook

  Now that the motivation and some syntax for exceptions has been presented, we can delve a bit deeper into the structure of writing programs containing exceptions. The following lesson is designed to help put into context the rudiments just presented. In addition, one more important component of exception handling is the 'raise' statement which is useful for raising an exception if, for example, it occurs inside an exception handler.

- More ExamplesPage

  This set of examples reviews the try, except, and raise commands. Make sure to practice them. All too often in introductory programming courses, the practice of handling errors, if taught at all, is brought up as an afterthought or some ancillary programming feature. Nothing can be further from the truth. To become a professional programmer, testing, debugging, and creating bulletproof software lies at the heart of software design. This is why alpha and beta versions exist. You can set yourself apart by refining your expertise in exception handling.

9.2: Handling a File Error

- Handling a File ErrorBook

  As already pointed out in the exceptions lesson, Python contains a vast set of built-in exceptions. One important class has to do with file handling. The example presented in the lesson will help to review file handling methods and put them in the context of handling exceptions.

Unit 10: Object-Oriented Programming

We are now ready to transition into object-oriented programming, which organizes code in the form of what are referred to as classes. In Python, every variable created is an object, and, as you have already seen, each variable has access to a set of methods. This is because there exists a class definition housing the methods that a given object has access to. In this unit, you will learn how to design your own classes, create or "instantiate" objects from a given class, and write programs that apply your class designs.

Completing this unit should take you approximately 8 hours.

- Upon successful completion of this unit, you will be able to:
  - explain the differences between procedural, structured, and object-oriented programming;
  - explain how classes, objects, and instances are used in object-oriented programming; and
  - implement simple programs that use classes, objects, and instances.

10.1: Overview of Object-Oriented Programming

Structured Programming and Procedural ProgrammingBook

We have been learning, accessing, and applying methods available in Python (it would be impossible to teach Python without doing so). Although the syntax and use of these methods are inherently object-oriented, we have been using them in the context of procedural program design and in the form of structured programs. Recall that procedural programs use functions as a way of breaking a program down into a set of procedures to solve a problem. Read this page to learn why we have been arranging our programs the way we have.

Procedural Programming versus Object-Oriented ProgrammingPage

It is important to understand how object-oriented programs differ from procedural programs. The main goal of object-oriented programming is to allow the problem being solved to dictate the class design. For instance, a house is organized as a function of its rooms: living room, kitchen, bedroom, and so on. Each room is organized as a function of its furniture and its use. Each room could define a new class containing attributes that define the furniture and methods that define their use. The process of breaking a problem down into a set of classes each with its own set of attributes and methods is called data abstraction. This process is in stark contrast to procedural programming. Read this page to see how these approaches differ.

10.2: Object-Oriented Programming (OOP)

Try typing these commands in Repl.it:

```
a=[1,2,3,4,5]
print(dir(a))
```

After creating the list object, you should see that the dir command generates a complete list of methods available for operating on that object. Notice that some of the methods have a double underscore on both sides of the method name. For instance, the `__add__` method has a double underscore as both a prefix and a suffix, while the append command does not. Methods that have a double underscore as both a prefix and a suffix are called "dunder methods" (for double underscore) or "magic methods". Magic methods do not need to be called explicitly. As we will see in the examples ahead, magic methods run automatically in response to some action invoked by some other command. The main point is that an object is somehow able to access a suite of methods. In this unit, you will learn how and why this is so. The first step is to master some basic terminology that will link together and reinforce terms and concepts relevant to object-oriented programming such as classes, methods, attributes, objects, instances, and so on.

- Basic TerminologyBook

Read this page to learn more.

- We will now introduce Python class construction along with how to create object instances. The example below defines a class `Rectangle` along with various methods useful for operating on rectangles such as `area` and `perimeter`. Run the following code in Repl.it:

```
class Rectangle:
  def __init__(self,x,y):
          self.length=x
          self.width=y
  def __str__(self):
          return 'Length='+str(self.length)+' Width='+str(self.width)
  def area(self):
          return self.width*self.length
  def perimeter(self):
          return 2*self.width+2*self.length
  def rescale(self,a):
          self.width=a*self.width
          self.length=a*self.length
  def __del__(self):
          print("This object no longer exists")
examp1=Rectangle(3,5)
print(type(examp1))
print(examp1)
print(examp1.area())
print(examp1.perimeter())
examp1.rescale(2)
print(examp1)
examp2=Rectangle(2,4)
print(examp2)
print(examp2.area())
print(examp2.perimeter())

examp1=42
del examp2
```

Some important observations:

- The keyword `class` alerts Python that a new class is being defined.
- The class name is Rectangle and the declaration terminates with a colon (:).
    - Generally, as a convention, the class name is capitalized.
    - All subsequent code contained within the class definition is indented.
- Method definitions (in a manner similar to functions) commence with the def keyword and terminate with a colon (:).
- `__init__`, `__del__`, and `__str__` are Python magic methods.
- `area`, `perimeter` and `rescale` are user-defined methods contained within the `Rectangle` class.
- The self variable is the object currently being referenced (more on this in a moment).
    - In Python, `self` is not a reserved keyword, but it is a very strong convention to use this variable name when referring to an object within a class definition.
- `examp1` and `examp2` are objects instantiated from the class `Rectangle`
    - The attributes for each object are `length` and `width`
    - Each object is initialized with its own set of attribute values
- The syntax of method calls such as `examp1.perimeter()` or `examp1.rescale(2)` using the dot notation should be familiar at this point in the course.

You can think of the class definition as a brand new variable type endowed with all the attributes and methods defined within the class. The code following the class definition demonstrates how it can be used.Given the class definition, it is possible to create multiple instances called "objects". Each object has access to the methods defined within the class, but retains its own attribute values. For instance, the commands

```
examp1=Rectangle(3,5)
examp2=Rectangle(2,4)
```

will instantiate two objects from the class `Rectangle: examp1` and `examp2`.Each object can use the class methods to compute the area, the perimeter, or rescale using the length and width provided as the input.

The `examp1` and `examp2` objects are said to be instances of the class `Rectangle`.At the time the`Rectangle(3,5)` and `Rectangle(2,4)`commands execute, the `__init__` method is automatically called. When `Rectangle(3,5)` executes, the length and width attributes for examp1 are set to the input values and, similarly for`examp2` when `Rectangle(2,4)` executes. Therefore, the `__init__` method is responsible for initializing the attribute values where `self` is the object being referred to at the time of execution (you may see others refer to

the `__init__` method as a constructor). Hence, it is possible to create multiple instances from the same class. This is the beauty of object-oriented programming: each instance has its own attribute values and can reference methods available within the class definition.

One more important point about methods internal to a class definition must be noted. Bear in mind that `self` refers to the object currently being operated upon. When a method is called to operate upon an object, observe that `self` is an implicit parameter that is not explicitly passed. For example, in the main code, `examp1=Rectangle(3,5)` instantiates an object by automatically calling the `__init__` method. Notice that there are two arguments passed in the Rectangle(3,5) statement. On the other hand, inside the class definition, the `__init__` method has three input arguments including self which must appear first on the input argument list. As another example, consider the `examp1.rescale(2)` command. In this case, the input parameter is the value used to rescale the length and width attribute values internal to the `examp1` object. Observe that, internal to the class definition, the rescale method has two input arguments. Again, the `self` argument is implicit and appears first while the scale parameter is explicitly set by the method call in the main code. Whenever a method is to be called to operate upon an object, the method definition must have `self` as the first input argument. After that, all subsequent input arguments should refer to parameters used within the method.

The `__del__` dunder method is referred to as the destructor and is responsible for deleting the object when its use has expired. The subject of destructors is often omitted in introductory courses. However, professional coding demands efficient memory management so that if an object is no longer being used, best practices require its removal from memory. In the above example, redefining `examp1` as an integer automatically causes the destructor to run as the object has been overwritten. The `del` command can be used to delete a variable; hence, the destructor would run upon deleting `examp2`.

The `__str__` method enables the `print(examp1)` command to make sense. Under normal operating circumstances, Python has no idea how to print objects. If no guidance is given, Python simply outputs a 'handle' that can be used to deduce the object's location in memory. This kind of information would not be useful to a high-level programmer. The `__str__` method basically gives guidance on how to apply the `print` command.

Magic methods are incredibly useful in this way because they execute in response to some action. The `__init__` method runs automatically when an object is instantiated, the `__del__` method automatically executes when an object is removed and the `__str__` method automatically executes when a desired string operation on an object is referred to.

To conclude, you now have a working knowledge of the following terms: class, object, instantiation, self, constructor, destructor, method, and magic method. In addition, you also know how to define a class and instantiate objects with respect to a class. Practice more examples to become comfortable with object-oriented programming.

- Creating Classes and MethodsBook

Read this for more on creating classes and methods.

- Magic MethodsPage

Python is packed with a vast set of magic methods. Before delving deeper into class constructions, it is a good idea to gain some more perspective on this subject.

- Going DeeperPage

This video demonstrates class construction with a balanced mix of magic methods and user-defined methods. You may need to refer back to the Basic Terminology section to understand how the `__add__` and `__str__` magic methods are being used.

- 10.3: Derived Classes
  - Classes and their associated methods are incredibly useful because of their reusability. Consider the case where we want to create a `Square` class that operates on squares. Given that the `Rectangle` class already exists, it should be possible for the `Square` class to inherit some of the computational capabilities of the `Rectangle` class presented in Section 10.2 since a square is a kind of rectangle. Whenever one class is a kind of another class, inheritance should be possible. This relationship is referred to as IS-A. Under these circumstances, a child class (e.g. `Square`) can inherit the use of various methods from the parent class (e.g. `Rectangle`).

Try inserting this class definition to the code from Section 10.2, which should follow directly after the `Rectangle` class definition.

```
class Square(Rectangle):
    def apratio(self):
        return .25*self.length
```

Then add this code to your main code:

```
examp3=Square(2,2)
print(examp3.area())
print(examp3.perimeter())
print(examp3.apratio())
```

There are several points to note regarding the syntax and usage of inherited classes:

The syntax class `Square(Rectangle)` means that `Square` is the child class and `Rectangle` is the parent class

`Square` has no `__init__` method. The object `examp3`, which is of the `Square` class, is allowed to use the `__init__` method from the `Rectangle` parent class. `Square` also inherits the ability to use other parent methods, such as `perimeter` and `area`. Study these examples to practice inheritance.

- InheritanceBook

Read this for more on inheritance.

- Going DeeperPage

The inheritance examples we've looked at so far have been short so that we can get the basic points across. Consider this example to practice working with a larger-scale object-oriented program.

- 10.4: Applying Object-Oriented Programming

- An Example of OOP and InheritancePage

This is an excellent example of boolean operations like AND, OR, and NOT. Following and implementing this example in Repl.it will help you review the OOP concepts in this unit.

https://learn.saylor.org/mod/book/view.php?id=30399&chapterid=6270