Python For Data Science

David Mukajanga

8/23/2021

Introduction

This course is taken as a second step toward learning data science. The goal here is to get started with the basics of the Python language and how it is used for data analysis. The course book is "Python For Everybody" as written by Dr. Charles Severance. Welcome!

Module 1: Introduction To Programming.

Contents:

1. What is Programming.
2. Why Learn Programming
3. Characteristics of Programming
4. Introduction To Python

By the end of this chapter, you are supposed to be conversant with the following terms:

- Constants
- Variables
- Types of variables
- Nimonik (Mnemonic) variable names
- Operators
- Operator precedence / Expressions
- Variable type conversion
- Commenting in Python
- Why program in the first place.

What is Programming?

The art of giving instructions to a computer hardware such that it is capable of executing a particular function.

Why Learn Programming?

- Most of the time is either you want to create a computerized tool that will enable you to carry out a specific function or you want to analyze a particular set of data.
- Most of those who use programming as a means to create digital tools aim at becoming software engineers. Some of them do this as their full time job while others like a hobby.
- Most of those who learn programming without wanting to become professional software developers are people like librarians, lawyers, doctors/academicians and all professionals that deal with data in their process of decision making.
- Sometimes they outsource this function but most of the time they learn the art themselves because it gives them a driver's seat in being able to do their jobs as they are supposed to be exactly done.
- For this course, the notes will base in data analysis. If you get so much intrigued so that you want to explore more advanced courses in programming, I, the writer of these notes wishes you all the best in your journey of learning.

Characteristics of Programming

1. Precision

- The difference between instructing a human versus instructing a computer is that a human can correct errors in the instructions as long as they account them.
- A computer on the other hand, will stop and surrender if it encounters anything it doesn't understand. This is commonly known as a "Syntax Error".
- Syntax Errors usually come from either mis-spelling the characters in the code (command), or mis-arrangement of words inside the command.
- For this reason, your code can be failing because of something as simple as putting one spacing instead of 2 in your code.
- So keep in mind to be precise when you are writing your codes.

2. Hardware

- The computer hardware is the one responsible for executing all your instructions and make possible for whatever task you had in mind to be executable.
- The next question to ask oneself now is how does it do that? Well the process goes like this:-
    1. You write your instruction to the **Secondary** memory. This is what is commonly known as a Hard disk. It includes all readable media such as flash drives, cd etc.
        - The instructions you write here stay here even if the computer is turned off.

- Sometimes the instructions come directly from input devices such as a gamepad. Most of the time the instructions will still go through your secondary memory because the main program lives there.

2. When you switch on your computer, your instructions are uploaded to the **Main** memory. This is what is commonly known as RAM.
   - Your instructions are translated from a computer language like Python to a machine language which binary, by using a program called **Compiler**. More on this later.

   - Once in the **main** memory, they stay there temporary while they are being worked on by the CPU and then they get erased once the program is shut down or the computer is shut down.

3. The **CPU** is ***just*** a very sophisticated hand calculator with the basic task of asking **-"What is Next!"-**. What it does is take your translated instructions, solve them, and return the answer back to the main memory.

4. The answer gets outputed back via the output devices such as a screen or speaker.

3. Intimidating / Full of Bluffs.

- Your main enemy will be "Syntax Error".

- At some point it will be frustrating but do not get discouraged. The word only means that Python (or any other language of programming you are using) is not understanding what you are saying. Be patient with yourself and do the work bit by bit. In the end you will become fluent in it.

Introduction to Python

What is Python?

- It is a programming language made in the late 1980's as an attempt to creating a powerful but easy to learn programming language.

- It was so because programming languages of that time were very technical and had a steep learning curve.

So with Python you should be able to hit the ground running in your computer programming journey.

Requirements

1. No previous programming knowledge needed.

2. A computer

3. A text editor for programming such as Notepad++, Atom etc. Atom is recommended because it behaves similarly in Windows, Mac and Linux computers. Get it **here**

4. Python itself. You can download it **here**.

Starting With Python

- The important thing to note before we continue is that we will be ***writing*** our code in the text editor, which in this case is Atom, and ***running*** that code through the *command prompt* in Windows computers and the *terminal* in Macs.

- There are ways in which the code can be run directly in the text editor but that is not very helpful for us now because there will be stages in which we will need to read data from different directories and it is good for us if we know how to work with the command prompt.

Important Terminologies.

1. Interactive Programming.

- This is when you are writing code and executing it in the same time.

- You usually do this when for example you are coding directly in the Windows or Mac's terminal.

- It is a little bit fun but it is not convenient in writing a huge chunk of coding lines in it since you lack most of the tools from programming text editors that enable you to write your code correctly and in the end your code is not saved.

2. The Python Interpreter (>>>) A.K.A *The Chevron Script*

- It is a sign that you can type and execute a python command.
- If this symbol also appears in the Window's Command Prompt or Mac's Terminal, it means that python has been opened and is running in the background.

- It also means you can program directly from the command prompt or terminal but your code will not be saved.

3. Errors.

1. ***Syntax Error*** = You should know this by now :) :).

2. ***ValueError*** = When you are trying to coerce a value of data to a class of data that it can not be coerced to. Eg, letters to integers.

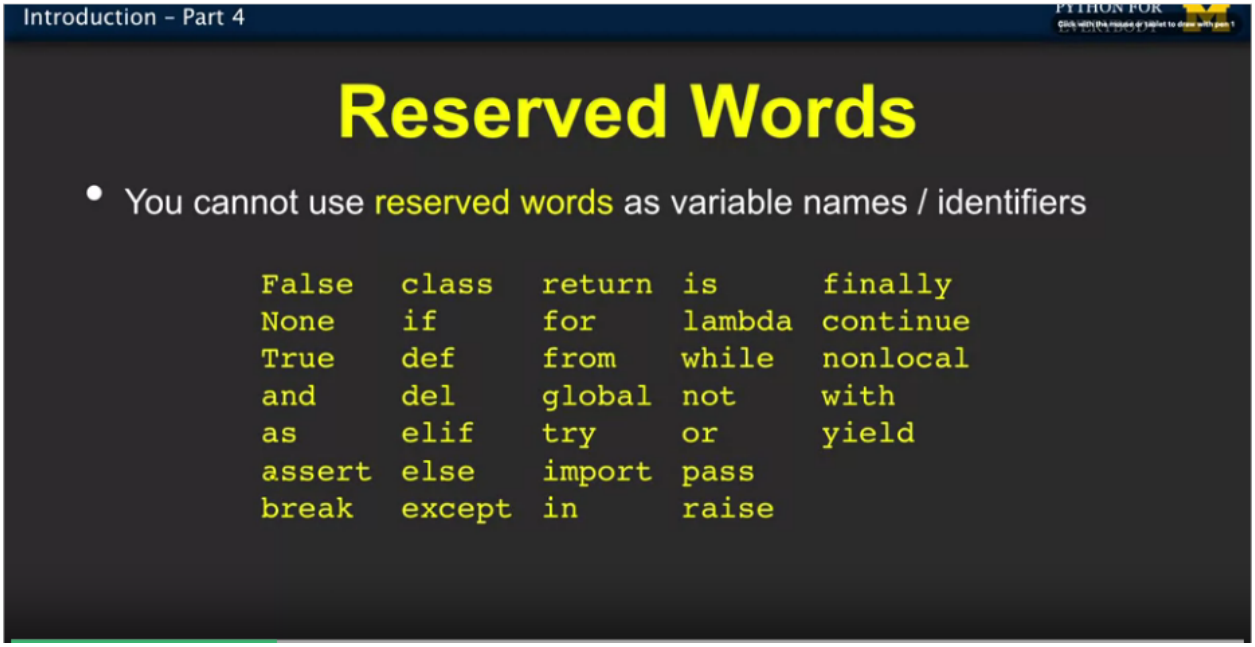
3. ***TypeError*** = When you are giving a mathematical relations between two different types of variables whose class of data don't relate mathematically. Eg. trying to sum a lettered variable and an integer variable.

4. Reserved Words

- These are function specific words that are used in Python to tell it to do a specific task.

- Under no situation can these words be used to mean something else than what the programming language understand them to be mean.

- This is the list of them:



5. Sentences / Lines

- Basically are lines of code
- Contains either the Assignment Statement (variable & its value), Assignment with an Expression (variable, its value + a command) or a function (eg. print(variable name goes here)).

6. Operators.

- These are symbols used to instruct the Programming language like Python to carry out a specific task.
- The following are common operators:
  - Assignment Operator ('=') = Used to assign a specific value to a variable. Eg, Variable1 = 2.
  - Addition = (+)
  - Subtraction = (-)
  - Multiplication = (*)
  - Division = (/)
  - Power = (**)
  - Remainder (%) = This publishes **just** the remainders in the division equation.
    Eg, if you input the following code:

```
>>> x = 9 / 2
>>> print(x)
x = 4.5
```

But if you input this one:

```
>>> x = 9 % 2
>>> print (x)
x = 5 #(Because 5 is the remainder in 4.5)
```

- Larger than / Smaller than = “>” “<”
- Equal to = “==” (Basically asks the question ‘Is... equal to ...?’)
- Larger than or Equal to = “>=”
- Smaller than or Equal to = “<=”

7. Expressions

- These are commands in the right side of the assignment operator .i.e. ‘=’.
- They should not be confused with constants in such a code as: x = **2**. Here 2 is a constant (also used as “value” in the previous explanations), not an expression.
- The following is an example of an expression:-

```
python       >>> VarX = 2  #(VarX is assigned a constant)       >>> VarA = VarX + 2
#(VarA is an expression here)
```

8. Variables.

- A.K.A variable names, these are numbers, letters or string of words which are used to represent something in the programming codes.
  - Eg. In this small code “x = 2”, x is a variable while 2 is a constant.
- The ***type*** of a variable depends on the ***class*** of data in that variable.
  - **String variables** are all variables whose data is enclosed by single quotes or double quotes .i.e. ‘data here’ / "" . In these type of variables, even if the data inside the quotes is numbers or has numbers, python will still read these as letters.
  - **Integers**. These are numbers that are presented only in a round form .i.e. 10 instead of 10.0 or 10.1 etc.
  - **Floating numbers (Real Numbers)**. These are the opposite of integers, that is, can be presented in fractions, decimals etc. Covers the whole range of numbers.
    N.B

- The difference between floating numbers and integers in Python is of great importance since some variables are supposed to only be presented in one class and not the other. Money for example should not be input in terms of floating numbers in python. That's just a rule of thumb.

- There is a way of *coercing* one type of variable to become another one, depending on the suitability of the class of data within that variable.
- For example, the '123' in code `>>> x = '123'` is a string variable and thus will be read as letters within python.

- So if we want to add 1, for example, by using the following code, the following error will occur in the execution of our code;

```
>>> x = '123'
>>> a = x + 1
>>> print(a)
TypeError: Can't convert 'int' object to str implicitly.
```

- To coerce this into an integer, the "**int**()" function is used. Eg:

```
>>> x = '123'
>>> y = int(x)
>>> a = x + 1
>>> print (a)
124.
```

- But not all variable types can be coerced in integers or floating numbers. (At least in python. In R it can be done.)

- In python, to use the **int()** or the **float()** function, there has to be numbers in that string for it to be transformed into an integer variable. Look at the following example:

```
>>> x = 'Hello Bob'
>>> a = int(x)
ValueError: Invalid literal for int() with base 10: 'x'
```

- To know the type of your variable, type the following function: type(Variable Name). Eg.

```
>>> x = '123'
>>> type(x)
<class 'int'>
```

- Integers can be coerced into floating numbers by using the "float()" function. Eg.

```
>>> x = 11
>>> float(x)
11.0
```

9. Mnenomic (Nimonik) Variable Names

- This is a system of naming variables in programming that makes the code easy to understand for a human being.
- For Example: The following variable names are completely fine for Python:

```
>>> x3ni21 = 2
>>> x3pi21 = 4
>>> x3pi12 = (x3ni21 * x3pi21) / x3pi21
```

- But a human reading this code will have a hard time telling the variables apart.
- This may lead to grave problems if they confuse one variable with another in debugging or a similar exercise.
- Instead, if variable names like the following are used, notice how it is now easier to digest the code:

```
>>> score = 2
>>> no.courses = 4
>>> gpa = (score * no.courses) / no.courses
```

- So Mnemonic naming is naming variables so that they can be clear, memorable & non-confusable.
- An important thing to keep in mind is to ***not*** use the 'reserved words' as variable names.

10. Scripts.

- These are text files containing all of the Python code.

Commenting in Python

- The symbol `#` is used for commenting in python.
- Essentially, everything that comes at the right side of that symbol in a python file is regarded as a comment and will thus be ignored in the execution of the code.
- This means that you can put instructions, comments and even reserve code under that symbol and the program will not look at it.
- This is useful in situations where you need to share your code to someone else or you are coming to read your own code after a while and don't remember why you did a particular function.
- It is therefore advisable to put instructions in the form of comments in your code in major steps so that it should be easy for you in the future.

Module 2: The Beginning: Learning To Swim In Python

Content

In this chapter, you will start to learn the basics of python through writing simple python programs. By the end of this course, you should have known the following:

- Likely sources of errors in your code
- The "Input()" Function.
- Purpose of the "Print()" Function
- Indentations and Code Blocks
- Sequential Codes, Conditional Codes and Repeating/Loop Codes

Likely Sources of Errors In your Code.

Before we begin, I am necessitated to bring the following things to your attention, things that made my first codes not work. They are small stuff which you too can make and they can make your brain heat up when you don't notice them as the source of problems in your code.

1. Capitalizing the command words..

- Don't capitalize command words such as "if", "elif" etc. Your code simply will not work, the main reason being the capitalized command words will be read as variables instead of as commands.
- The sign for this is the fact that the command words won't be colored if you are using a programming text editor like Atom. If you notice that the command words are of the same color as the variable names then know that you didn't write them correctly.
- Don't even capitalize the first letter. Python wants to read the command statement like an "if" as "if ..." not "If..."

2. Not Coercing The User-Input into Integers or Real Numbers When Using Comparison Operators Against Numerical Values.

- Look at the following example:

```
>>> xscr = input("Please Enter Your Score: ")
>>> if xscr > 75:
        print ("You can enroll to the course")
```

- The code above will show a ***TypeError*** in the second line telling you this:

  `TypeError: '>' not supported between instances of 'str' and 'int'. .`

- This is because, at the second line, we are comparing the original value of the variable `xscr` which is still read as a string (even if the user inserted a number), to 75 which is a numeric. To solve this, the code should be re-written like so:

```
>>> xscr = ***float***(input("Please Enter Your Score: "))
>>> if xscr > 75:
        print ("You can enroll to the course")
```

- Notice now that the value that will be compared to 75 is a floated numerical value of the variable `xscr` from the first line.

3. Putting an Instruction In Front of the "else" Statement.

- Look at the following code:

```
>>> xscr = float(input("Please Enter Your Score: "))
>>> if xscr >75 :
      print('You can enroll to the course.')
>>> elif xscr > 50 :
        print('You have to take the preparation course.')
>>> else xscr < 50:
        print('You have to retake the entrance exam.')
```

- The code will give the following error in the third line, at the 'else' statement:
`SyntaxError: invalid systax`
- This is because, although the programmer had a good intention of giving very specific instructions to python, python already knows that if you use the "else" statement, you already made up your mind that anything else that is different from the 'if' or 'elif' statements you made earlier should be executed by the else statement.
- So, to avoid this problem, write your else statements like so:

```
>>> xscr = float(input("Please Enter Your Score: "))
>>> if Xscr > 75:
        print("You can enroll to the course")
>>> elif xscr > 50:
        print ("You have to take the preparation course")
>>> else :
      print ("You have to repeat the entrance exam")
```

Post your problem in online forums.

- Sometimes the only people that can see the problem in your code are anybody else but you :) :).
- This is usually because they are more experts or experienced than you :) :).
- Post your code in online developer communities like ***Stack Overflow***, ***Coursera Community*** and ***Cross Validated*** to get help.
- Sometimes the problem might be as complicated as having used different ***character codes*** in your code.

- The following is the guideline for asking questions in the forums:

1. Let other people know that you have done all the previous necessary steps.

2. Read the documentation :) :), just be patient and read it.

3. Describe the goal. This is more important than describing steps.

4. Be explicit about your question.

5. Provide as minimum amount of information as it possibly appropriate.

6. Follow-up with the answer if/when you find it.

Program 1: Pay Calculator.

The "input("")" Function

- It is a programming function that instruct the program to pause and ask for a user input before using that input to continue executing the written code.

- As it can be noticed, the inside of the brackets in the input function has double quotient marks which means that everything that gets entered into that function is a **STRING** data. This includes numbers.

- Therefore, in an example like this, let us write a program to prompt a user for hours and rate per hour in order to calculate the gross pay:

```
>>> hrs = input("Enter Hours:")
>>> rate = input("Enter Rate:")
>>> pay = float(hrs) * float(rate)
>>> print ("Pay:", pay)
```

In this code:

1. The user will be asked to input the number of hours worked through the first line. The words "Enter Hours:" are the ones that will be displayed in her screen. The input from the answer will be regarded as a string regardless of whether the answer is in numbers or text. In order to thus be able to force the answer into numbers, the programmer has to allow only a numerical input in the interface of the user.

2. The second line will get executed and the user will be asked to enter the amount of rate per hour. The same applies here.

3. The third variable is assigned the coerced values of var1 & var2

4. The output of the third variable is displayed to the user with the words "Pay:" preceding the value of the third variable. The output will look like this: - **Pay: 96.25**

The "print()" Function

- This displays the output of the code to the user of the program.

- Several ways of using this is covered as we move forward so pay attention to the explanations on how it was used in the coming codes.

Program 2: The Course Admission Checker

The Python Flow

- This is the order in which Python executes the written code.
- This order is largely dependent on **indentation**.

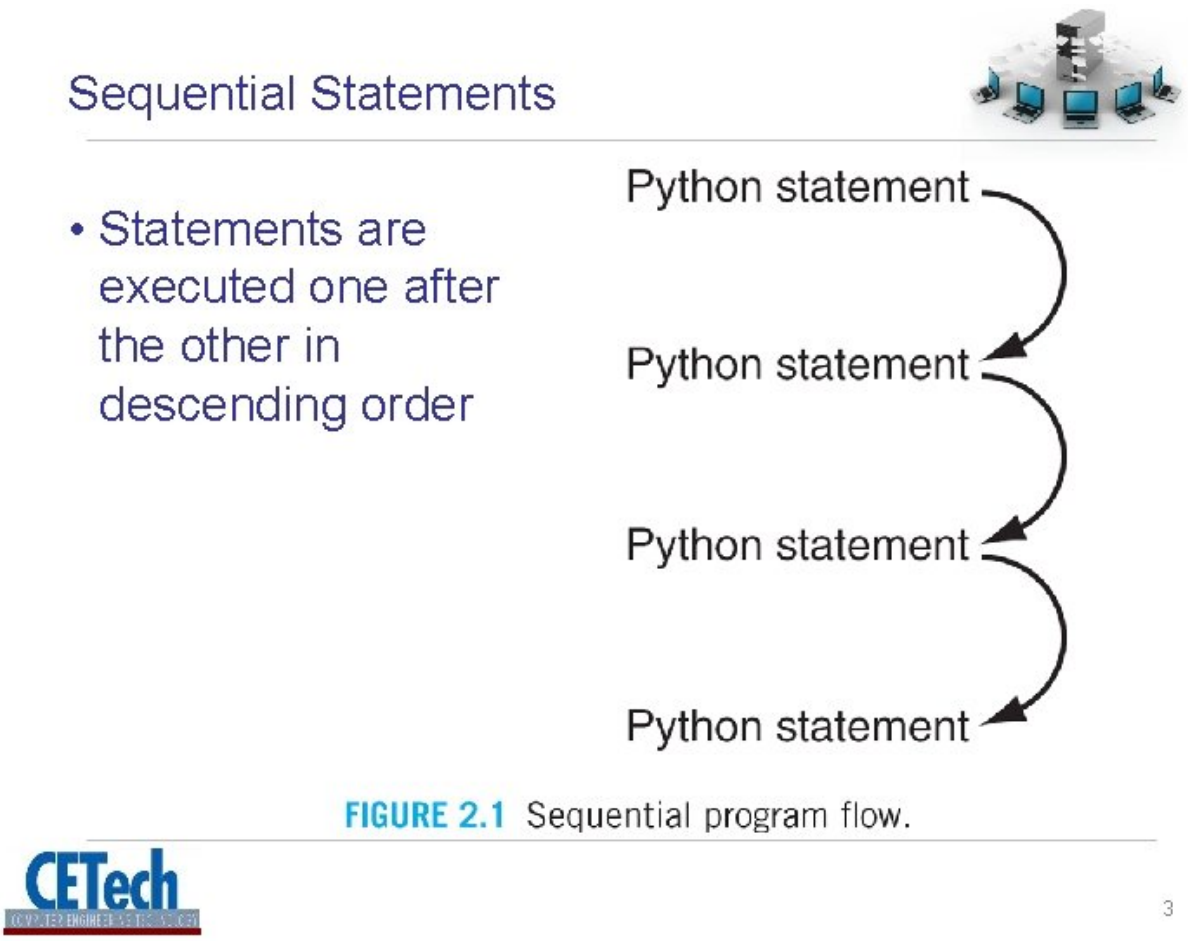
Indentation

- In python, indentation gives a signal to which code will be run first.
- Indentation comes as a result of the type of python code/statement written. Because of this, a part of code that has several indented lines in it is called a "Code Block" or "Code Chunk" because it gets executed as one line of code as per instructions written in it.
- Different from other languages like C++ where indentation will not cause your code not to work, in Python, indentation can be a source of your code not being able to run.
- The main cause is usually mixing the indentation by using both the 'TAB' key and the 'SPACE' key to make indentation in the same document. The space created by these two keys are interpreted differently in the program, thus creating an error in execution.
- Indentation made by using the 'SPACE' key is known as **Soft Tab** while the one made by the 'TAB' key is known as the **Hard Tab**.
- It is a good practice to configure your text editor so that it uses soft tabs even when you press the 'TAB' key.
- To do this, in your text editor, go to: File > Setting > Editor > Tab Type. There are usually 3 options there, you should choose soft.
- The procedure might differ depending on the text editor you are using so if you are having a hard time finding those setting, searching on Google or YouTube might help.

Types of Python Statements/Codes

Sequential Statements/Codes

- These are python codes written in such a manner that they are executed in a sequence from the first line to the last.
- It is the default execution pattern in python. All the other execution flows are embedded in a sequential execution of the code.
- The picture of the following chart flow explains this clearly:



Conditional Statements

- These are python codes that will not get executed unless a particular condition specified by the programmer is met.
- The use of conditional statements in programming is also known as the use of "**Logic**".
- As the name suggests, **Logic** as used in programming is the common sense that you put in how a computer program executes the task you give it to execute.
- Conditional statements are the building blocks of logic since they are the ones' giving a computer program a sort of rudimentary intelligence on what to do and when.
- They are made by using an "**if**" statement and further clarified by using the "**elif**" or "**else**" statements.

1. The "if" Statements

- Consider the following example:
  Eg.

```
>>> x = float(Input("Please Enter Your Score Here: "))
>>> if X > 75:
```

```
            print ('You can enroll in the program,please choose the starting day that is most appro
>>> if x < 75:
            print ('Please repeat the course')
>>> if x > 75:
            input("Enter The Date Here: ")
```

- In this code, if the user's score is above 75, the second line of code with the instructions "Please repeat the course" will not be executed. If the grade is smaller that 75 then the third line of code with the instructions "Enter The Date Here:") will not be executed.

2. The "elif" and "else" statements

- The word "elif" is a combination of "else" and "if".

- This statement is used to give more clarification on what should be done if the first condition created by using the "if" statement was not met.

- Consider the following example:

```
>>> xscr = float(input("Please Enter Your Score: "))
>>> if xscr > 75:
        print("You can enroll to the course")
>>> elif xscr > 50:
        print("You have to take the preparation course")
>>> else :
        print ("You have to repeat the entrance exam")
```

- Like it can be observed, we have basically instructed the program to tell our user to take the preparation course if our user enters any value less than 75 but more than 50 by using the ***elif*** statement.

- In this example, if our user indeed enters a value less than 75 but more than 50, the first code under the "If" statement will not be executed and the program will move on to the second instruction which is the "elif" statement.

- If that doesn't work too it will move to the 'else' statement.

- In short, anywhere where the three conditional statements are put together, only one condition will be executed.
- If, for example, we didn't put any clarification of what should be done if the value entered is less than 75, then the program will not do anything as it won't execute the "If" line and there is nothing else after that.

Store and Re-use Statements...

- They are used to store a block of code that might later be needed to be executed somewhere else.
- These are made by using the "def" statement. "def" here means 'define'.

- A 'def' statement is known as a "function" because it is later used as a function to call the execution of the code stored in it.
- Take the following example:

```
>>> def computepay(h, r):
        if h <= 40:
            print (h * r)
        elif h > 40 :
            return ((40*r) + ((h - 40)*(r * 1.5)))

>>> try :
       h = (float(input("Please Enter Hours: ")))
       r = (float(input("Please Enter Rate: ")))
>>> except :
     print("Please Enter Numeric Values Only")
     h = (float(input("Please Enter Hours: ")))
     r = (float(input("Please Enter Rate: ")))
>>> print ("Pay:", computepay(h, r))
```

- In the code above:
  1. "computepay()" = *Function Name*
  2. (h, r) = *Parameters*

Parameters

- These are variable names that we put into the 'def' statement to represent user input.
- They can sometimes be variables that only exist in the 'def' statement. When they are used in this manner, the goal is usually to assign ***arguments*** to them.
- They also help us programmers understand what is going on in the 'def' statement. Check the following example:

```
>>> def greeting(lang) :
        if lang == 'en' :
            print ('Hello!')
        elif lang == 'es' :
            print('Hola!')
        elif lang == 'sw' :
            print ('Habari!')

>>> greeting('sw') :
      print ('greeting()', Bob)
>>> greeting ('es') :
      print ('greeting()', Juma)
>>> greeting ('en') :
      print ('greeting()', Sarah)
```

- In the code above:

  - 'lang' = Parameter
  - 'en', 'sw', 'es' = Arguments
- The thing to note is the following:
  - When we first run the program, the whole 'def' block will be skipped. Python will only take note that there is a function called 'greeting' in that block of code and that whenever that function gets mentioned later in the code it should come back and execute the code therein.

Return statements...

- This is a statement used to give an answer to the defined 'def' statement.
- It is used in a situation where the 'def' function has been given its arguments, has done the computation needed and has to ***return*** a value to be used as the value of the function call in the ***calling expression***.
- For example:
  -> def greet() return "Hello"
  -> print(greet(), "Glenn") -> print(greet(), "Sally")
- Here whenever the *greet()* function has been invoked, the answer that will be returned is "*Hello*" OR

```
-> def greet(lang):
       if lang == 'es':
          ***return*** 'Hola'
       elif lang == 'fr':
          ***return*** 'Bonjour'
       else:
          ***return*** 'Hello'
-> print(greet('en'), 'Glenn')
Hello Glenn
-> print (greet('es'), 'Sally')
Hola Sally
-> print(greet('fr'). 'Michael')
Bonjour Michael.
```

- 'def' functions that have to return an answer after it has executed its function is called a ***fruitful function***.
- ***Unfruitful Functions*** are 'def' statements that do not have a return value.
- These usually are the 'def' statements which have been ordered to execute a specific task without the need to provide a value in the end.
- Eg.

```
>>> def end() :
       print('Thanks to the Almighty you are all well. Have a good time.')
       quit()
```

```
>>> x = input('Hey! Welcome. I am a virtual doctor appointment secretary. Do you need an appointmen
>>> if x == 'Yes'
        y = input('Ok. Select your most appropriate date from the list of available dates below: ')
>>> elif x == 'No'
        end()
```

- The 'def' statement here just closes the program. It does not return any value.

Repeating/Loop Statements.

- There are made by either using the 'while' statement or the 'for' statement.

1. Indefinite Loops

- These are made by using the `while` statement.
- They essentially repeat the same thing until a specific condition/logic created by the programmer is met.
- The condition is known as the "iteration statement".
- The iteration statement is usually placed in the 'while' statement line and what python essentially does with the statement is asking whether the iteration statement is true or false. If true, it executes the instructions in the 'while' block. If not, it skips to the next line of code.
- Consider the following example:

```
>>> x = 5
>>> while x > 0 :
        print (x)
        x = x - 1

>>> print('Blast off')
```

- In the code above, the iteration statement is `x > 0`.
- As long as the assigned value of x is greater than zero, python will continue to run the code in the indented portion of the while statement. Only when `x == 0` will python exit that code block and continue downwards.
- The output of the above code will be as following:

```
C:\Users\David\Desktop\DS\Programs>the_while_statement.py
5
4
3
2
1
Blast off

C:\Users\David\Desktop\DS\Programs>
```

1.1. Infinite Loops

- These are 'while' statements with either no iteration statement or an iteration statement which always remains true.

- Consider the following example :

```
>>> x = 5
>>> while x > 0 :
        print (x - 1)

>>> print (Blast off!)
```

- In this code, the iteration statement is always true. Python will print number 4 forever because:

  1. Python will see that `x` has been assigned a value of 5.

  2. It would have been told that as long as the value of x is greater than 0, it is supposed to be printing (x - 1) (that's how the 'while' statement functions).

  3. When it finishes to print (x - 1), it goes back to evaluate the iteration statement in the 'while' statement. The problem is that we have not put any instruction that changes the value of x and thus the value of 'x' in this block will always remain greater than zero and thus the while statement will run without stopping until the computer is shut down.

- Therefore, it is important to make sure that the iteration statement written in a loop statement has an end.

1.2. The 'break' statement.

- This is used to put another iteration statement within the 'while' statement.

- It basically tells python to "break" repetition and ***exit*** the 'while' statement if that second condition is met.

- Consider the following example :

```
>>> x = input('Do you need an appointment (y/n): ')
>>> while x == 'y' :
        y = input("Please select doctors' office numbers below: ")
        if (y == '3') or (y == '5') or (y == '7') :
            print('Your appointment has been accepted, please fill out your information')
            break
        else :
            print('This doctor is on a lunch break, please select another number')

>>> pname = input('Enter your full name please: ')
>>> quit ()
```

- In the code above, once the patient chooses 3, the 'else' statement won't be executed and instead python will "break" out of the while statement and move on to execute the variable name "pname".

1.3. The 'continue' statement.

- The 'continue' statement basically tells python to stop where it is at in the 'while' or 'for' statement and start again from the beginning of it.
- Like the 'break' statement, it also contains a specific condition that if met, the statement will instruct python to repeat the cycle of iteration by "continuing" to run the 'while' or 'for' statement.
- Take the following example:

```
>>> while True:
>>>   line = input ('> ')
      if line[0] == '#' :
          continue
      if line == 'done'
          break
      print (line)
>>> print('Done')
```

- In the code above, any input that starts with a "#" sign will not be printed and python will stop the iteration from there and start it from above, not executing any statements below the continue statement in the code block.
- So for example if we set empty answers in the survey to be "#", that line will be skipped whenever this code is run and instead python will go back up and ask the input as instructed in the variable "*line*"
-

2. The Definite Loops

- These are made by using the `for` statement
- Usually used for values that are predefined by the programmer.
- This includes things as lists or lines in a file or characters in a string.

- In the `for` statement, the iteration variable is assigned the particular values in the order they occur.
- This is done by the usage of the `in` statement along with the for statement.
- Consider the following example:
- Take the following example:

```
>>> for var1 in [5, 4, 3, 2, 1] :
        print (var1)
>>> print ('Blastoff!')
```

- The output will be the same as the previous one `(i.e. 5, 4, 3, 2, 1, Blastoff!)`, but with this code, the machine knows which number starts/ends and in which order.
- The 'for' statement is made more straightforward in that it is told specifically which values a variable will take and in which order.
- It is like an automated version of the 'while' statement, but one that is carefully made to outline what does what. :) :).

Loop Idioms

- It is another way of saying; ***Situations which loops are usually used for***.
- They include the following:

1. Finding The Largest / The Smallest Value In The Loop.

- Can be done by using both the `for` and `while` statement, depending on the situation.
- The `for` statement is used when there is a particular set of values already outlined in the program.
- The `while` becomes useful when the values are not specifically outlined, like when the values are coming directly from the user input.
- Because a computation like this is directly on the go, `while` is better because we don't really know how many values will be entered.
  - Consider the following example:

```
biggest = None
smallest = None
while True :
    usx = input('Enter a number: ').lower()
    if usx == 'done' :
        break
    try :
```

```
            int_usx = int(usx)
            if biggest is None :
                biggest = int_usx
            elif biggest < int_usx :
                biggest = int_usx
            if smallest is None :
                smallest = int_usx
            elif smallest > int_usx :
                smallest = int_usx
            else :
                continue
        except :
            print('Invalid input')

    print('Maximum is', biggest)
    print('Minimum is', smallest)
```

- In this code:

    1. There are "**comparison/initial**" variables. These are `biggest` and `smallest` and they are used to set the initial benchmark for the initial value of the loop.

    2. Iteration variable is set as `True`. This means that as long as the logical conditions we have put inside the loop statement are met, this loop will continue to run. The `True/False` statements are mainly used in indefinite loops.

- Notice the use of `is` line 9 and 13.

- `is` and `is not` are operators used to show absolute accuracy whether something `is` or `is not` the same as something else.

- Notice also the usage of the `None` variable.

- This variable is used to assign an empty space to a variable of choice.

- It is useful in these kinds of situations for enabling the comparison of the values in the loop against themselves so that an accurate answer of which one is the biggest/smallest can be obtained.

- If, for example, the programmer is finding a smallest number from a user input and sets the value of the comparison variable to a particular number, say 0, the output of the program will come out false if the user enters numbers that are all greater than zero since the smallest number that will be produced by the program will be zero instead of the actual smallest number of the user input.

- Consider the example below for a specified list, using the `for` statement.

```
>>> init = None
>>> for largest in (9, 54, 39, 12, 23, 43):
        if init == None :
            init = Largest
        elif largest > Init :
            init = Largest
        else :
```

```
                continue

    >>> print (init)
```

The code above also uses the `None` variable to establish a logic that:
1. The first value in the list should be compared to an empty value from the comparison variable (`init`).
2. Be the value of the comparison variable.
3. Then be the value which the other values are compared to and being replaced by any value that will be less than it in the list.

2. Looping Through a Set.

- This is basically when you instruct the program to print every value that is in the loop.
- This is useful when you have a certain set of values that are either generated by the program or being begotten somewhere and you want to display them to the user.
- For example, the following code will print numbers from 0 to 10..

```
for it in range(0, 11)
    print(it)
```

- The `range()` function is used here to set the boundaries of what we want to print. The formula for using the `range()` is like this: `range(start value, stop value [, step (how many numbers you want to skip per iteration) value])`. Be sure to put the comma signs as they have been placed in the formula.

3. Counting, adding and finding the average in the values of a Loop

- To count in a loop, we introduce a counting ***initial*** variable assigned to start at zero and add 1 to itself every time the loop runs.
- To add, we do the same thing with the initial comparison variable and we instruct it to add itself to the value of the iteration variable every time the repetition takes place.
- To find an average, we divide the comparison variable for summation over the comparison variable for counting.
- By using these two comparison variables; .i.e. the one for addition and the one for counting, many other statistical operations can be derived from them.
- Consider the following example:

```
count = 0
tot = 0
for x in range(11, 41) :
    count = count + 1
    tot = tot + x
```

```
        mean = tot/count
        deviation = (x - mean)
        w = (deviation)**2
        totdev = 0
        totdev = totdev + w
        totdev = totdev/count
        def std(rsqdev)
            rsqdev = (totdev)**0.5

    print('Count =', count, 'Mean =', mean, 'Standard Deviation = ', std(rsqdev))
```

- The above code uses the `for` loop to count the number of values available, sum them and eventually find mean and standard deviation.

4. Filtering in a loop.

- Can be done for multitude of purposes.
- Maybe certain numbers from a range of numbers need to be shown to the user, filtering can be done to achieve this.
- Or a programmer might be in need of certain numbers in a loop, filtering can be used to catch these numbers as the loop runs.
- Loops are set by using the `if` statement within the loop.
- Consider the following example:

```
    for x in range(3, 43)
        if x == (13), (14), (17) :
            print(x)
```

5. Searching in a Loop.

- This is done by using the **Boolean** variable.
- This variable has only two values: `True` / `False`.
- The values `True` or `False` should be entered with the first letter being **capital letter**. Otherwise it will not be read by python.
- It is usually used with the `while` and `if` statements, to basically instruct python that as long as the certain piece of code/logic is "True / False", to continue to run the loop.
- To search by using the Boolean variable in a loop, the initial comparison variable should be set to `False` and put an instruction in the loop that whenever the value we want is found, this initial variable should be changed to `True`.

- Consider the following example:

```
init_var = False
for x in range(3, 10) :
    print(x)
    if x == 5 :
        init_var = True
        print('Found', init_var)
```

Strings in Python

Basic Features of String Variables

- Can be concatenated (joined) together by using the "+" operator.
- This is to remind you that the "+" operator serves a different purpose on strings compared to numbers.
- The "+" operator cannot be used between numerical and string variables.
- Take the following example:

```
>>> a = 'String'
>>> numerical = 10
>>> print ('I just merged a $' + a + 'with a $' + numerical 'variable'.)
TypeError: must be str, not int
```

Operations That Are Done On Strings.

- There are several things that can be done to text in python. The following are among the major operations done to strings in python.

1. Indexing The Letters in a Block of Text.

- By default, python assigns a number to each letter of a string variable.
- The numbers starts with zero in the first letter and ascends to the length of the text.
- Indexing is calling out a specific letter or group of letters in python by using the assigned numbers.
- This is done by using the square brackets "`[]`", with the *index*(the number of the letter), inside the brackets.
- The brackets are put in the end of the string variable.
- Consider the following example:

```
>>> ex = 'Example'
>>> ind = ex[0], ex[1], ex[2], ex[3]
>>> print(ind)

('E', 'x', 'a', 'm')
```

2. Looping Through a String

- Can be used with an indexing variable.

```
>>> ex = 'Example'
>>> ind = ex[0], ex[1], ex[2], ex[3]
>>> for letter in ind :
        print(letter)
E
x
a
m
```

3. Counting the Number of Letters in a String.

- This is done by using the `len()` function.
- The variable name of the string variable in question goes into the brackets.
- Consider the following example:

```
>>> name = 'David, Mukajanga'
>>> count = len(name)
>>> print(count)
16
```

- The code above counted all the letter, including all white spaces and all the commas.

4. Slicing

5. Search & Replace

6. Stripping White Space

String Library

- A library in Python is a list of all functions that can be used to on different types of variables, eg., string variables have their own libraries, numerical variables their own etc.
-

Prefixes

Parsing and Extracting

Module 3: Moving Beyond Shallow Waters

Contents

Reading Files into Python

Components of the Computer You've Been Utilizing So Far..

Opening A File

File Handles..

File Structure

Newline Character

Reading a File Using the `For` Loop

Searching For Lines

Reading The File Names

Dealing With Bad File Names

Data Structures

Difference Btn Algorithms and Data Structures

Data Structures 1: Lists

- This is the organized version of collections.

Concept of a Collection

Lists and Definite Loops

Indexing and Lookup

Mutability of Lists

Function: Len, Min, Max, Sum

Slicing Lists

List Methods: Append, Remove

Creating An Empty List

Sorting Out Lists

The Split Fuction

Splitting Strings Into Lists of Words

Using Split to Parse Strings

Data Structure 2: Tuples

Features of Tuples

Tuples in Assignment Statements

Immutability of Tuples

Comparability of Tuples

Sorting Tuples

Data Structures 3: Dictionaries

- An unorganized version of collections in python.
- It is a list-like accumulation of values that are given specific labels.
- The labels are used to identify the value later in the program.
- Consider the following example:

```
>>> purse = {}
>>> purse['PaperMoney'] = 12
>>> purse ['Bank Card'] = 14
>>> purse['Coins'] = 13
>>> purse['Bus Card'] = 39
>>> print(purse)
{'Money': 12, 'key': 14, 'perfume': 13, 'Bus Card': 39}
```

- From the code above, a wallet has been taken to exemplify what a python dictionary is.

    1. A new empty dictionary has been made. This is done by assigning empty curly brackets (`{}`) or using the `dict()` function on a new variable.

    2. We added values and assigned keys to them.

    3. Values and Keys can be added like ***this***;

    ```
    variableName = {'Key': Value, 'Key': Value}
    ```

    or like this:

    ```
    VariableName = {}
    VariableName['key'] = Value
    VariableName['key'] = Value
    ```

    So in our code above, values are 12, with key "Money", 14 "Bank Card", 13 "Coins" and 39 "Bus Card".

- So, in a normal wallet, colors and textures would be used to identify the value of paper money / a coin. In python the "Keys" are used for that purpose.

Functions used in Dictionaries

Looping Through Dictionaries

Common Applications of Dictionaries

Counting The Frequency

Debugging

The First Step

- This involves putting a print statement just above the line that has traceback.
- This helps to understand what the program was doing before it encountered the error in the problematic line.
- Sometimes the error is not in the line itself but rather the logic that is put on the code.
- Printing out the results before the traceback helps to know whether it is the line or the logic that is faulty.
- Take the following example:

The Guardian Pattern

The Short-Circut Evaluation

- This is the practice of using a compound guardian statement.
- A Compound Guardian Statement is a guardian statement that has two or more conditions in it.
- It is made by using the `if` and `or` statements in the following manner:

```
if (first condition goes here) or (second condition goes here) :
    # Do something
```

- Conditions in this statement are executed in sequence, .i.e, if the first condition is met, the second condition will not be run.
- This is the **most important** thing to know about the short-circuit evaluation.
- It is also the main reason why it is used in the first place.
- Take the following example:

```
att = ['Gvt Officials', 'Organizers', 'Students', 'Public', '99']
for num in att :
    if
```

Regular Expressions