**INTRODUCTION TO PYTHON**

**WEEK: PYTHON SYNTAX**

**DAY 1: INTRODUCTION TO PYTHON**

**PYTHON IN THE JOB MARKET**  
  
Python is constantly growing in the market. With Python knowledge, you may find yourself in top positions within a few years, but you should increase your knowledge as much as possible. You will find that you often need a working knowledge of other programming languages, such as the language courses provided by CapaCiTi, to be qualified for a position in a company. This course covers important topics that you need to grasp the foundation of Python and make you ready for a position in the job market.  
  
There are various job fields that Python has to offer. Integrating applications with MySQL is in high demand because MySQL and Python are both open-source applications. This means that some companies are switching over from their current expensive systems to open-source systems. YouTube and BitTorrent are examples of companies that use Python.  
  
Network programming in Python is another option, which requires an extensive knowledge of how networking is controlled. Another fields for a Python programmer are Software Engineer, Software developer, Research Analyst, Data Analyst and Data Scientist; you will often be required to have database experience when working in this field.  
  
This course is aimed at teaching you as much as possible in a relatively short time; its aim is to extend your programming knowledge further and introduce you to a new programming syntax.

**Installing Anaconda on Windows**  
  
To install Anaconda on Windows, follow these steps:  
  
1. \*\*Download Anaconda\*\*:  
- Go to the Anaconda download page: https://www.anaconda.com/products/distribution.  
- Choose the version you want to install (either Anaconda or Miniconda) based on your requirements. Anaconda is a larger distribution with many pre-installed packages, while Miniconda is a minimal installer that allows you to install packages as needed. For most users, Anaconda is a good choice.  
- Download the Windows installer corresponding to your system architecture (usually 64-bit).  
  
2. \*\*Run the Installer\*\*:  
- Locate the downloaded installer file, usually named something like "Anaconda3-<version>-Windows-x86\_64.exe" and double-click it to run the installer.  
  
3. \*\*Install Anaconda\*\*:  
- The Anaconda installer will open. Click "Next" on the welcome screen.  
  
4. \*\*Read and Accept the License Agreement\*\*:  
- Carefully read the license agreement and select "I Agree" if you accept the terms.  
  
5. \*\*Select Installation Type\*\*:  
- Choose whether you want to install Anaconda for just your user (recommended) or for all users on the system. It's generally a good practice to install it only for your user unless you have a specific reason to install it system-wide.  
  
6. \*\*Choose Installation Location\*\*:  
- Choose the location where Anaconda will be installed. The default location should be fine for most users.  
  
7. \*\*Advanced Options (Optional)\*\*:  
- You can add Anaconda to your system PATH if you want. This makes it easier to use Anaconda from the Command Prompt. Leave this option checked.  
  
8. \*\*Install Microsoft VSCode (Optional)\*\*:  
- Anaconda Navigator offers the option to install Microsoft Visual Studio Code. You can choose to install it or not based on your preference.  
  
9. \*\*Install Anaconda Navigator (Optional)\*\*:  
- Navigator is a user-friendly graphical interface for managing packages and environments. It's recommended to install it, so leave the option checked.  
  
10. \*\*Install Anaconda 3\*\*:  
- Click the "Install" button to start the installation process. This may take a few minutes.  
  
11. \*\*Complete Installation\*\*:  
- After the installation is complete, click "Next" and then "Finish" to exit the installer.  
  
12. \*\*Start Anaconda Navigator\*\*:  
- You can start Anaconda Navigator by searching for it in your Windows start menu.  
  
13. \*\*Using Anaconda\*\*:  
- Once Anaconda Navigator is open, you can use it to create and manage Python environments and install packages. You can also use Anaconda Prompt to work with your environments and packages through the command line.  
  
That's it! You've successfully installed Anaconda on your Windows system. You can now create Python environments, install packages, and start coding in Python.

**History of Python**  
  
Python was conceived in the late 1980s and Guido van Rossum started implementing it at CWI in the Netherlands in December 1989. It is a relatively simple language that includes a standard library that provides modules for a large number of processes that programs deal with. This approach keeps Python simple yet reliable programming language.  
  
Python has an easy-to-use syntax that is focused on the programmer who must type in the program, read what was typed, and provide formal documentation for the program. Many languages have syntax focused on developing a simple, fast compiler; but those languages may sacrifice readability and can be more difficult to write. Python strikes a good balance between fast compilations and readability, and it is easier to write applications.  
  
Python is implemented in C and relies on the extensive, well understood, portable C libraries. It fits seamlessly with UNIX, Linux, and POSIX environments. Since these standard C libraries are widely available for the various MS-Windows variants, and other non-POSIX operating systems, Python runs similarly in all environments. The Python programming language was created based on lessons learned during language and operating system support. Python is built from concepts in the ABC and Modula-3 languages.

**Interactive Mode**  
  
Interactive mode in Python allows you to enter and execute Python statements and expressions one at a time, immediately seeing the results. This is particularly useful for quick testing, experimentation, and learning. You can access the Python interactive mode in the following ways:  
  
1. \*\*Python Interactive Shell (Python REPL)\*\*:  
- Open your command prompt or terminal.  
- Type `python` and press Enter. This will start the Python interactive shell.  
- You'll see a prompt, usually `>>>`, where you can enter Python commands.  
- For example:  
```python  
>>> print("Hello, world!")  
Hello, world!  
>>> 2 + 3  
5  
>>> exit() # To exit the Python interactive shell  
```  
  
2. \*\*IPython\*\* (Interactive Python):  
- IPython is an enhanced interactive Python shell that provides additional features and improvements over the default Python REPL.  
- To use IPython, you need to install it (if not already installed) using `pip` or `conda`. Here's how you can start IPython:  
- Open your command prompt or terminal.  
- Type `ipython` and press Enter.  
- You'll see an enhanced prompt that allows you to use features like tab-completion and interactive help.  
  
3. \*\*Jupyter Notebook\*\*:  
- Jupyter Notebook is an interactive web-based environment that allows you to create and share documents containing live code, equations, visualizations, and narrative text. It's particularly popular for data science and data analysis tasks.  
- To use Jupyter Notebook, you need to install it using `pip` or `conda`. Once installed, you can start a Jupyter Notebook server by running the following command:  
```  
jupyter notebook  
```  
- This will open a web browser with the Jupyter interface, where you can create new notebooks and execute code cells interactively.  
  
Interactive mode is a great way to experiment with Python code, test small code snippets, and quickly verify the behavior of functions and libraries. It's an essential tool for learning Python and for interactive development and debugging.

**Comments in Python**  
  
In programming, comments are a programming language construct used to insert human-readable text in the source code of a program. These extra pieces of text are ignored by the compiler and interpreter but can be potentially significant to programmers. Comments are added to make the source code easier to understand.  
  
Comments could be used for a wide range of purposes, for example:  
  
Augmenting program code with basic descriptions to generate external documentation.  
Integration with source code management systems and other kinds of external programming tools.  
Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

**Using Python**  
  
Python is a versatile and popular programming language that can be used for a wide range of tasks, from web development and data analysis to scientific computing and automation. To use Python effectively, you need to follow these steps:  
  
1. \*\*Install Python\*\*:  
- If Python is not already installed on your computer, you can download it from the official Python website (https://www.python.org/downloads/) and follow the installation instructions.  
  
2. \*\*Choose a Development Environment\*\*:  
- You can write Python code in various development environments. Some popular options include:  
- \*\*IDLE\*\*: This is the default integrated development environment that comes with Python.  
- \*\*Text Editors\*\*: You can use simple text editors like Notepad (Windows) or TextEdit (macOS) for writing Python scripts.  
- \*\*Integrated Development Environments (IDEs)\*\*: There are several Python-specific IDEs available, such as PyCharm, Visual Studio Code, and Jupyter Notebook.  
- \*\*Jupyter Notebooks\*\*: Particularly useful for data analysis and interactive coding, Jupyter notebooks provide a web-based interactive environment.  
  
3. \*\*Write Python Code\*\*:  
- Open your chosen development environment and start writing Python code. A simple "Hello, World!" example would look like this:  
```python  
print("Hello, World!")  
```  
  
4. \*\*Running Python Code\*\*:  
- To run your Python code, you can either:  
- Save your code in a `.py` file and run it from the command line using `python filename.py`.  
- If you're using an interactive environment like IDLE, IDEs, or Jupyter, you can execute code interactively within the environment.  
  
5. \*\*Learn Python\*\*:  
- Python has a rich ecosystem of libraries and frameworks for different purposes. Depending on your goals, you might want to learn:  
- Basic Python syntax and data structures (e.g., variables, lists, loops, functions).  
- Python libraries for specific tasks (e.g., NumPy for numerical computing, pandas for data analysis, Flask/Django for web development).  
- Object-oriented programming (OOP) in Python.  
- Error handling and debugging techniques.  
- Best practices and code style (e.g., using PEP 8 guidelines).  
  
6. \*\*Explore Python Ecosystem\*\*:  
- Python has a vast ecosystem of libraries and packages. You can use `pip` (Python's package manager) to install additional packages as needed. For example, to install the `requests` library, use:  
```  
pip install requests  
```  
  
7. \*\*Practice and Projects\*\*:  
- The best way to learn Python is by practicing and working on projects. Try to build small applications or scripts to reinforce your knowledge and skills.  
  
8. \*\*Documentation and Resources\*\*:  
- Refer to Python's official documentation (https://docs.python.org/3/) for detailed information on Python's features and standard libraries.  
- There are many online tutorials, courses, and books available to help you learn Python. Websites like Python.org, RealPython, and Python.org's Beginner's Guide are excellent starting points.  
  
9. \*\*Online Communities\*\*:  
- Join online Python communities, such as Stack Overflow, Reddit's r/learnpython, or other forums, to ask questions, seek help, and learn from others.  
  
10. \*\*Keep Learning\*\*:  
- Python is a dynamic language with a vibrant community. Continuously learn and explore new libraries and techniques to expand your Python skills.  
  
Whether you're a beginner or an experienced developer, Python can be a valuable tool for various applications. It's known for its simplicity and readability, making it an excellent choice for both newcomers and seasoned programmers.

**Strings**

Strings are a fundamental data type in Python and many other programming languages. They are used to represent text or sequences of characters. In Python, strings are typically enclosed in single (' '), double (" "), or triple (''' ' ') quotes.

**DAY 2: VARIABLES**

**Introduction to Variables**  
  
Variables are a temporary storage space in a computer’s memory. When a variable’s value changes the program’s current state also changes. A variable acts as a container to hold a different number of data items or values. All programming languages use variables, as they are among the most important elements in programming, and that is why a good understanding of variables will only make your job easier when writing programs. Variables are also used to move data between functions; this will be discussed later.  
  
Every variable is created with an initial value. A variable can be in three states:  
  
Variable creation (Declaration)  
Variable assignment (Initialization)  
Variable changed (Execution)  
Once the code which created the variable has finished executing, the variable is destroyed.  
  
In Python, variables are defined in a standard way, by using the assignment character (=). This changes the value of the variable. Naming conventions specify the way in which variables should be named. This standard is used to make code more readable, and thus easier to understand.  
  
The rules include the start and continuation characters. Variable names may contain any upper or lower case letter (A–Z, a–z), a number, or the underscore character. They may not begin with a number or contain spaces. Continuation characters are any characters except whitespace characters like tab and space.  
  
Here are a few examples of valid variable names:  
  
c  
ref\_number  
admin  
aVeryLongName  
Here are a few examples of invalid variable names:  
  
True  
$name  
12Graph  
In Python identifiers are case sensitive, so for example, firstName, FirstName, FIRSTNAME, and firstname are four different identifiers. A second rule is that variables cannot have the same name as Python’s keywords. We can find out what keywords are in Python, by using the function called dir(). If this function is called with the \_\_builtins\_\_ attribute, it returns a list of Python’s built-in attributes.  
  
The \_\_builtins\_\_ module contains all Python’s built-in attributes, which can be used with the dir()function. The ones that are returned are identified with the following characteristics:  
  
Python’s built-in exceptions start with a capital letter.  
The rest are either functions or data type names.  
Identifiers that start and end with one or two underscores are special methods.  
NOTE All of the methods, exceptions, and functions contained in  
dir(\_\_builtins\_\_), dir(\_\_doc\_\_), dir(\_\_name\_\_), and dir(\_\_package\_\_) cannot be used as variable names.

**Using variables**  
  
All variables have to be assigned to a data type like a string (a series of characters) or an integer (positive and negative whole numbers). There are others, some of which will be discussed at a later stage.  
  
Python has a powerful feature regarding the assignment. A variable is assigned automatically to an appropriate data type. For example, Python automatically assigns a variable to a string data-type, if an input or value is given that contains letters or words. Values of the same type can be manipulated together. Sometimes Python finds a way to manipulate values into a common type by casting the values automatically. There are cases where values need to be cast explicitly. The example code below illustrates how values are assigned to variables automatically:  
  
Example 1: Automatic assignment of variables:  
  
  
num1 = 15 #num1 is automatically assigned as an integer  
  
#value  
  
num2 = 30 #num2 is also assigned as an integer value  
  
print (num1 + num2) #answer is printed  
  
  
  
  
Output:  
  
  
>>>  
  
45  
  
>>>  
  
  
  
  
This previous example adds two integer values. This will not result in an error as both num1 and num2 are integers. In other cases, things would not run so smoothly. For instance, if an integer value were to be added to a string value an error would occur.

**Casting**  
  
Casting can be done in two ways:  
  
Implicitly: The compiler automatically casts a value from one data type to another when assured that there will be no data loss.  
For example. casting from an integer variable to a floating-point variable or casting from an integer variable to another integer variable  
Explicitly: A value cannot be automatically cast from one data type to another if it will result in data loss. Extra code has to be written to ensure that the value stays the same and only the data type changes.  
For example, casting from a floating-point value to an integer value  
You often have to convert the values you input, in order to have a correct output. The following example illustrates this.  
  
Example 2: Explicit string and integer casting:  
  
  
num1 = 15 #num1 variable is automatically assigned  
  
#as an integer value  
  
num2 = "30" #num2 variable is automatically assigned  
  
#as a string value  
  
  
ans = num1 + num2 #ans is assigned to num1 added to num2  
  
  
print ("answer:", ans) #ans is printed  
  
  
  
The previous example will raise an error and will not execute at all. This is because num1 and num2 are not defined using the same data types or data types that can be cast automatically: num1 is an integer data type and num2 is a string data type. This means that and will not know if it should be assigned to an integer or a string. Either num1 or num2 should be cast to match the other one’s data type. The solution below will clarify any misunderstanding.  
  
Solution:  
  
  
num1 = 15 #num1 variable is automatically assigned  
  
#to an integer value  
  
num2 = "30" #num2 variable is automatically assigned  
  
#to a string value  
  
ans = num1 + int(num2) #num2 is cast to integer type and added #to num1  
  
print ( "num2 cast to integer ",ans)  
  
  
ans = str(num1) + num2 #num1 is cast to a string type  
  
  
print( "num1 cast to string ",ans) #to num1  
  
print ("num2 cast to integer: ", ans)  
  
  
ans = str(num1) + num2 #num1 is cast to string and num2 gets  
  
#added  
  
print ("num1 cast to string:", ans)  
  
  
  
  
Output:  
  
  
>>>  
  
num2 cast to integer: 45  
  
num1 cast to string: 1530  
  
num2 cast to integer: 1530  
  
num1 cast to string: 1530  
  
>>>  
  
  
  
  
Notice that the two print statements have different outputs. The first print statement prints the integer values added together (15 + 30 = 45). The second print statement prints the characters of num1 and num2 concatenated/joined together (‘1’ + ‘5’ + ‘3’ +’0’ = ’1530 ’). The difference between using ‘+’ or ‘,’:  
  
  
>>> #interpreter  
  
>>> print ("Gau","teng")  
  
Gau teng  
  
>>> print ("Gau"+"teng")  
  
Gauteng  
  
>>>  
  
  
  
  
The following example illustrates how Python’s interpreter automatically casts two values and adds them together:  
  
Example 3: Casting values:  
  
  
num1 = 15.60 #num1 variable is a floating point  
  
#number  
  
num2, num3 = 30, 32 #num2 variable is a integer  
  
  
ans = num1 + num2 + num3 #converted to floating point  
  
print ("The answer:", ans) #ans is printed  
  
  
  
  
Output:  
  
  
>>>  
  
The answer: 77.6  
  
>>>  
  
  
  
  
The previous piece of code will execute with no errors because it is possible to cast an integer to a floating-point without any data loss.  
  
Example 4: Casting values:  
  
  
i\_number = int(15)  
  
s\_number = str("12.543")  
  
s\_characters = str("five")  
  
f\_number = float(123.5675)  
  
  
f = float(i\_number)  
  
print ("An Integer cast as a floating point number:", f)  
  
  
i = int(f\_number)  
  
print ("A floating point number cast as an Integer:", i)  
  
  
f = float(s\_number)  
  
print ("A number string cast as a floating point number:", f)  
  
  
f = float(s\_characters)  
  
print ("A number string cast as a floating point number:", f)  
  
  
  
  
Output:  
  
  
>>>  
  
An Integer cast as a floating-point number: 15.0  
  
A floating-point number cast as an Integer: 123  
  
A number string cast as a floating point number: 12.543  
  
Traceback (most recent call last):  
  
File "C:\float\_casting", line 15, in <module>  
  
f = float(s\_characters)  
  
ValueError: could not convert string to float: five  
  
>>>  
  
  
  
  
Notice that no errors occur when converting from an integer to a float. This will always be the case because an integer can be cast to a float data type implicitly because there will be no data loss. However the reverse is not true; a floating-point number cannot be implicitly cast into an integer as this will result in data loss (all data after the precision (.) will be lost), because the floating-point value does not get rounded off to the nearest whole number when implicitly converting to an integer (as seen in the above example).  
  
The third printed line casts a string to a float successfully, because s\_number is in the correct format. The rules to convert a string to a float are:  
  
The string should only contain numbers.  
Other than numbers the following are allowed:  
Only one dot (.) character. Indicates the decimal starts after the dot (.) character.  
A ‘+’ or ‘−‘ character at the beginning of the string. This indicates that the number is either positive or negative.

**Defining Functions**

In Python, a function is a reusable block of code that performs a specific task or set of tasks. Functions are used to break down a program into smaller, manageable pieces, making the code more organized and easier to maintain. Here's how to define a function in Python:

```python

def function\_name(parameters):

# Function body

# Code to perform the desired task

# ...

return result # Optional

```

Let's break down the components of a function definition:

1. \*\*`def`\*\*: This keyword is used to define a function in Python. It is followed by the function's name.

2. \*\*`function\_name`\*\*: This is the name you give to your function. It should follow the same naming conventions as variable names (usually lowercase with words separated by underscores). It should also be descriptive of the function's purpose.

3. \*\*`parameters`\*\*: These are input values that the function accepts. Parameters are placed in parentheses and separated by commas. They are optional; a function can have no parameters or multiple parameters.

4. \*\*Function Body\*\*: This is where you write the code that performs the task or tasks the function is designed for. The body is indented to indicate that it is part of the function.

5. \*\*`return` (optional)\*\*: You can use the `return` statement to specify the value or values that the function should return when it's called. If a function doesn't have a `return` statement, it implicitly returns `None`.

Here's a simple example of a Python function that calculates the square of a number:

```python

def square(number):

result = number \*\* 2

return result

```

You can call this function and use it like this:

```python

result = square(5)

print(result) # Output: 25

```

This is just a basic example. Functions in Python can be more complex and can involve control structures, loops, and interactions with other parts of your program. They're a powerful way to structure your code and make it more modular and reusable.

**DAY 3: DATA TYPES**

**Data Types**  
  
Three data types have already been used in the examples before. Now we shall have a more detailed look at the data types used in Python. Data types have different functions in a program. One of the most important factors to take into consideration is that the performance of a computer can be severely affected by choosing the wrong data type.  
  
For example, a program might only need to use numbers between 1 and 20; it would therefore not make sense to use string variables to store these numbers in, as one will need to convert these strings back to numbers in order to perform calculations on them. This would create extra work for the computer, which means that the performance could degrade in larger programs. Therefore, one should rather define the variables as integers.  
  
Data types covered in this course include:  
  
Integers  
Booleans  
Floating point numbers  
Complex numbers  
Strings  
Literals are an alternative to using variables. Examples of literals include:  
  
"This is only a string"  
"\t"  
2  
Literals are values that never change, i.e. they are a constant. Look at the following example:  
  
  
>>> #Interpreter  
  
>>> print ("The answer: ", 5 + 2)  
  
>>> The answer: 7  
  
>>>  
  
  
  
5 and 2 are constant values; they are not assigned to a variable, and therefore are literal values.  
  
NOTE: You do not have to convert 5 + 2 to a string because the two integer values are added and automatically converted because explicit casting is not required in this case.  
  
Data Types  
  
Data Types Examples  
Integers These represent numbers in an unlimited range. This is only limited by a machine’s memory.  
Booleans Evaluate to ‘True or False’, 1 or 0 respectively.  
Floating point numbers Floating-point numbers represent double-precision numbers.  
Complex numbers Complex numbers represent numbers as a pair of double-precision numbers.  
Strings A sequence of Unicode characters e.g. a word or a sentence that can be manipulated.

**Integers**  
  
Python has three distinctive numeric types; this means that for every number that you use in your programs there is a suitable data type that you will use to ensure that your programs are truly performance efficient. As you know by now, integers are numbers. Numbers can have negative and positive values. Monetary numbers can have decimal values.  
  
We say that Python is a static language because only values of a certain type can be assigned to particular variables, according to their data types assigned.  
  
Integers are always whole numbers. Integers include negative and positive numbers. The only factor that determines the range of an integer variable is the amount of memory a machine has available.  
  
Manipulate integers  
  
Syntax Description  
a + b Adds number a and number b  
a - b Subtracts b from a  
a \* b Multiplies a with b  
a / b Divides a by b (always returns a floating-point number)  
  
NOTE: 5/5 will return a float value of 1; this value will be cast implicitly to an integer value that can be contained in an integer-type variable without losing any precision or data.  
  
The four main operators used when doing calculations, are ‘+’, ‘-’, ‘\*’, and ‘/’. There are more, but those will be discussed later in the course. The plus ‘+’ operator can be used universally, which means that you can use it for purposes other than to adding numbers together. The plus operator can also be used to concatenate strings.  
  
Up until now, programs had fixed values that were used to get an output by manipulating the given data. In real life situations, almost all programs require some sort of input from the user to give accurate output. This is done by using the input() command.  
  
The following examples illustrate how the different operators can be used to manipulate data entered by the user:  
  
A company called BIG VEGGIE sells fresh food to smaller stores. BIG VEGGIE buys their stock in bulk. BIG VEGGIE has asked you to develop a system to calculate the amount small stores will have to pay. All stock items have a fixed price:  
  
Asparagus – R30.54 per kg  
Beetroot – R1.45 per kg  
Broccoli – R14.43 per kg  
Garlic – R35.81 per kg  
Potatoes – R2.04 per kg  
A store must buy more than 100 kg of broccoli in order to get a 20% discount. They must also buy more than 300 kg of potatoes in order to get a 30% discount. BIG VEGGIE has asked you to develop a program that will only ask them to input the weight for each product the store buys from them. The program must determine the total amount due. Presume you have to take more than 100 kg of broccoli, and more than 300 kg of potatoes. The rest of the products have a fixed price, independent of how much you buy.  
  
Example 1: Fresh food company:  
  
NOTE: From now on, the coding examples in this course include line numbers for explanatory purposes, but numbering should not be included when typing the code in your editor.  
  
  
1 Asparagus = 30.54  
  
2 Beetroot = 1.45  
  
3 Broccoli = 14.43  
  
4 Garlic = 35.81  
  
5 Potatoes = 2.04  
  
6  
  
7 print ("Enter the amount of Asparagus (kg):")  
  
8 aspa = float(input())  
  
9 print ("Enter the amount of Beetroot (kg):")  
  
10 beet = float(input())  
  
11 print ("Enter the amount of Broccoli (kg):")  
  
12 broc = float(input())  
  
13 print ("Enter the amount of Garlic (kg):")  
  
14 garl = float(input())  
  
15 print ("Enter the amount of Potatoes (kg):")  
  
16 pota = float(input())  
  
17  
  
18 aspa = aspa \* Asparagus  
  
19 beet = beet \* Beetroot  
  
20 b = (((broc \* Broccoli) \* 20) / 100)  
  
21 broc = (broc \* Broccoli) - b  
  
22 garl = garl \* Garlic  
  
23 p = (((pota \* Potatoes) \* 30) / 100)  
  
24 pota = (pota \* Potatoes) - p  
  
25  
  
26 total = aspa + beet + broc + garl + pota  
  
27 print ("The total the store has to pay: R", total)  
  
  
  
  
Lines 1 to 5 set the price value of the vegetables that BIG VEGGIE has to offer; these prices are fixed. Lines 7 to 16 ask a user how much of each vegetable the store would like to buy (measured in kilograms). Lines 18 to 24 calculate the total price for each of the vegetables. Line 20 calculates how much discount should be given for broccoli, and line 21 then subtracts the discount from the total amount charged for broccoli. The same occurs on lines 23 and 24. Line 26 calculates the total that the store will have to pay. Line 27 prints the total.  
  
Output:  
  
  
>>>  
  
Enter the amount of Asparagus (kg):  
  
12  
  
Enter the amount of Beetroot (kg):  
  
32  
  
Enter the amount of Broccoli (kg):  
  
103  
  
Enter the amount of Garlic (kg):  
  
45  
  
Enter the amount of Potatoes (kg):  
  
417  
  
The total the store has to pay: R3808.838  
  
>>>

**Floating point numbers**  
  
Floating point numbers are better known as floats. Float is the data type that manages numbers with decimal places with very accurate precision. The float data type can be called as a function with zero or 1 argument of any data type. If no argument is given, then float returns 0.0. If an argument is given, an attempt will then be made to convert the value to a float data type, but this does not mean it is always possible. For example, float("21.765") will be converted to a float, but float("FF909A") will raise an exception.  
  
A string value cast to a float must contain only numbers and only one occurrence of the dot (.) character. The following piece of code will clarify the statement:  
  
The following example shows the different ways to format float numbers:  
  
Example 3: Format strings:  
  
  
>>> #Interpreter  
  
>>> print ("Today's Dollar price compared to the Rand: R%f" % 6.85871)  
  
Today's Dollar price compared to the Rand: R6.858710  
  
>>>  
  
>>> print ("Today's Dollar price compared to the Rand: R%.2f" % 6.85871)  
  
Today's Dollar price compared to the Rand: R6.86  
  
>>>  
  
>>> print ("Change since yesterday: R%+.2f" % 0.5)  
  
Change since yesterday: R+0.50  
  
>>>  
  
  
  
The example compares the American dollar to the South African rand over two days; the formats of the relevant values are changed to only print two decimal values.  
  
The f in the format part of the print statement indicates that the number to be formatted is a float.  
  
The + sign indicates that the changed amount must be signed, which means that if the dollar compared to the rand decreases, that a − sign would be printed before the amount difference.  
  
The following examples should clear any misunderstanding:  
  
  
>>> #Interpreter  
  
>>> print ("Negative amount: %+f" % -0.23)  
  
Negative amount: -0.230000  
  
>>> print ("Positive amount: %+f" % 0.23)  
  
Positive amount: +0.230000  
  
  
  
The following table shows tokens that can be used to format a string:  
  
Manipulate strings  
  
image 3  
  
The following example will use some of the tokens that Python uses:  
  
Example 4: Format strings:  
  
  
>>> #Interpreter  
  
>>> print ("18 (decimal) as a octal value: %o" % 18)  
  
18 (decimal) as an octal value: 22  
  
>>> print ("45 (decimal) as a uppercase (X) hexadecimal value: %X"% 45)  
  
45 (decimal) as an uppercase (X) hexadecimal value: 2D  
  
>>> print("4.6 (float): (e) floating point exponential value: %e"% 4.6)  
  
4.6 (float): (e) floating point exponential value: 4.600000e+00  
  
>>>

**Unpacking Argument Lists**

 STRINGS  
  
You have already used strings in some of the previous examples. Now you will be shown what strings actually are. Strings are represented by the immutable (unchangeable) str data type. Strings are a sequence of Unicode characters which form a single manageable string. The str data type can be called to create a string; when there is no argument supplied, it returns an empty string. s = str("") is the same as s = str(), when an argument is passed to the string method that is not a string value, it is passed as a string representation of the type supplied: s = str(17.2354), is the same as s = str("17.2354"). The string function is often used to convert other data types to strings.  
  
Example 6: Converting to string:  
  
  
1 print ("Please enter your name:")  
  
2 s\_name = input(str())  
  
3 print ("Please enter your surname:")  
  
4 s\_surname = input(str())  
  
5 print ("Please enter your current age")  
  
6 i\_age = input(int())  
  
7  
  
8 s = str(i\_age)  
  
9  
  
10 sentence = s\_name + " "  
  
11 sentence += s\_surname + " is currently "  
  
12 sentence += s + " year(s) old"  
  
13  
  
14 print (sentence)  
  
  
  
This example is straightforward. The user is asked to enter three values: his or her name, surname, and current age. Then a message is printed with the person’s name and surname and age with a single print statement. Look at the output of the code below if the program seems unclear.  
  
Output:  
  
  
>>>  
  
Please enter your name:  
  
John  
  
Please enter your surname:  
  
Doe  
  
Please enter your current age  
  
034  
  
John Doe is currently 34 year(s) old  
  
>>>  
  
The example above asks you to input information with the intention that the program will use the information to produce a suitable output. You are asked to enter your name first. Your name and surname are stored as objects of the string type.  
  
The ‘0’ in front of 34 in the output indicates that an integer value is expected.  
  
Using the ‘+=‘ operator with strings  
The ‘+=‘ operator adds values to an existing variable. This is illustrated in the next example:  
  
Example 7: The ‘+=‘ operator with strings:  
  
  
sentence = "This sentence is way too long to fit on one line of"  
  
sentence += " code and that is why I'm breaking this sentence down"  
  
sentence += " into pieces."  
  
  
print (sentence)  
  
  
  
The ‘+=‘ operator’s functionality is not only limited to strings, it can be used with other data types, too. More operators will be covered on Day 4.  
  
Using the end of line escape sequence  
The end of line escape sequence (\) can also help to make code more readable, as the following example shows:  
  
Example 8: End of line escape sequence:  
  
  
>>> #Interpreter  
  
>>> sentence = "This sentence is way too long to fit on one line of" \  
  
" code, and that’s why I'm breaking this sentence up" \  
  
" into pieces."  
  
  
>>> print (sentence)  
  
  
  
  
Enclose the expression in brackets  
The easiest way to write code in an easily readable format is by enclosing the expression in brackets:  
  
Example 9: Enclose the expression in brackets:  
  
  
sentence = ("This sentence is way too long to fit on one line of"  
  
" code, and that’s why I'm breaking this sentence up"  
  
" into pieces.")  
  
  
print (sentence)  
  
  
  
There are various ways to span a line of code over several lines. This is to make code easier to interpret and to fit long lines of code into the width of a page when printing is done.

**Lambda Expressions**  
  
Lambda expressions, also known as lambda functions or anonymous functions, are a feature in Python that allows you to create small, anonymous functions without the need to define a formal function using the `def` keyword. Lambda expressions are often used when you need a simple, short function for a specific purpose, especially when you need to pass a function as an argument to another function.  
  
The basic syntax of a lambda expression is as follows:  
  
```python  
lambda arguments: expression  
```  
  
Here's what each part of the lambda expression means:  
  
- `lambda`: The keyword used to define a lambda function.  
- `arguments`: The input parameters or arguments that the lambda function takes. These can be zero or more arguments separated by commas, similar to function arguments in a `def` statement.  
- `expression`: The single expression that is executed when the lambda function is called. The result of this expression is returned as the function's result.  
  
Lambda expressions are typically used with higher-order functions like `map()`, `filter()`, and `sorted()`, as well as when defining simple functions on the fly. Here are some examples to illustrate the use of lambda expressions:  
  
1. \*\*Simple Lambda Expression\*\*:  
```python  
square = lambda x: x\*\*2  
print(square(5)) # Output: 25  
```  
  
2. \*\*Using Lambda with `map()`\*\*:  
```python  
numbers = [1, 2, 3, 4, 5]  
squared\_numbers = list(map(lambda x: x\*\*2, numbers))  
print(squared\_numbers) # Output: [1, 4, 9, 16, 25]  
```  
  
3. \*\*Using Lambda with `filter()`\*\*:  
```python  
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))  
print(even\_numbers) # Output: [2, 4, 6, 8, 10]  
```  
  
4. \*\*Using Lambda for Sorting\*\*:  
```python  
names = ['Alice', 'Bob', 'Charlie', 'David', 'Eve']  
sorted\_names = sorted(names, key=lambda x: len(x))  
print(sorted\_names) # Output: ['Bob', 'Alice', 'David', 'Eve', 'Charlie']  
```  
  
Lambda expressions are a concise way to define small functions for specific tasks. However, they are not suitable for complex functions or those that require multiple statements. In such cases, it's better to use regular functions defined with the `def` keyword.

**Daily Notes - Conventions about the content and formatting of documentation strings**

 Conventions about the content and formatting of documentation strings  
  
Documentation strings, often referred to as "docstrings," are a way to document Python functions, classes, modules, and methods. Properly formatted docstrings are essential for making your code more understandable, maintainable, and accessible to other developers. Python has established conventions for the content and formatting of docstrings, which are commonly referred to as PEP 257. Here are some of the key conventions:  
  
1. \*\*Triple-Quoted Strings\*\*: Docstrings are typically enclosed in triple-quoted strings. You can use either double or single quotes, but triple-quotes are common because they can span multiple lines.  
  
2. \*\*One-Line Summary\*\*: A docstring should begin with a one-line summary of the function, class, or module's purpose. This summary should be concise and not exceed 72 characters in length.  
  
```python  
def my\_function():  
"""  
This is the one-line summary.  
"""  
```  
  
3. \*\*Multi-Line Description\*\*: Following the one-line summary, you can provide a more detailed description of the function, class, or module. This description should explain what the entity does, its inputs, outputs, and any other relevant information. The description should be indented and wrapped at 72 characters.  
  
```python  
def my\_function():  
"""  
This is the one-line summary.  
  
This is a more detailed description of what the function does.  
It can span multiple lines and should be wrapped at 72 characters.  
"""  
```  
  
4. \*\*Parameters and Return Values\*\*: For functions and methods, include a section that documents the parameters and return values. List each parameter, its data type, and a brief description. If a function returns a value, document it as well.  
  
```python  
def my\_function(param1, param2):  
"""  
This is the one-line summary.  
  
:param param1: Description of param1.  
:type param1: data\_type  
  
:param param2: Description of param2.  
:type param2: data\_type  
  
:return: Description of the return value.  
:rtype: data\_type  
"""  
```  
  
5. \*\*Module Docstrings\*\*: For modules, provide an overview of the module's purpose and contents, including any important classes, functions, or variables. This should appear at the top of the module file.  
  
```python  
"""  
This is a module-level docstring.  
  
It provides an overview of the module's purpose and contents.  
Include any important classes, functions, or variables here.  
"""  
```  
  
6. \*\*Class Docstrings\*\*: For classes, follow a similar structure as functions, but include class-specific details, including attributes and methods.  
  
```python  
class MyClass:  
"""  
This is a one-line summary of the class.  
  
:param attribute1: Description of attribute1.  
:type attribute1: data\_type  
  
:param attribute2: Description of attribute2.  
:type attribute2: data\_type  
"""  
  
def my\_method(self):  
"""  
This is a one-line summary of the method.  
  
:return: Description of the return value.  
:rtype: data\_type  
"""  
```  
  
7. \*\*Consistency\*\*: Consistency in docstring style is crucial for readability and maintainability. Adhering to common conventions and formatting helps ensure that docstrings are easy to understand and navigate.  
  
By following these conventions for docstrings, you can make your Python code more accessible to others and improve its maintainability. Additionally, tools like Sphinx can generate documentation from well-formatted docstrings, making it easier to create documentation for your code.

**DAY 4: OPERATORS**

**Introduction to operators**

**OPERATORS**  
  
In programming, operators are symbols or special keywords that are used to perform operations on data or variables. Operators are essential for performing mathematical, logical, and relational operations, making comparisons, and modifying data. Python, like many other programming languages, provides a variety of operators that can be classified into different categories. Here's an introduction to some common types of operators in Python:  
  
1. \*\*Arithmetic Operators\*\*:  
- Arithmetic operators are used for mathematical calculations like addition, subtraction, multiplication, division, modulus, and exponentiation.  
- Examples:  
- `+` (Addition)  
- `-` (Subtraction)  
- `\*` (Multiplication)  
- `/` (Division)  
- `%` (Modulus)  
- `\*\*` (Exponentiation)  
  
2. \*\*Comparison Operators\*\*:  
- Comparison operators are used to compare two values and return a Boolean result (True or False) based on the comparison.  
- Examples:  
- `==` (Equal to)  
- `!=` (Not equal to)  
- `<` (Less than)  
- `>` (Greater than)  
- `<=` (Less than or equal to)  
- `>=` (Greater than or equal to)  
  
3. \*\*Logical Operators\*\*:  
- Logical operators are used to combine or manipulate Boolean values. They are often used to make decisions or evaluate conditions.  
- Examples:  
- `and` (Logical AND)  
- `or` (Logical OR)  
- `not` (Logical NOT)  
  
4. \*\*Assignment Operators\*\*:  
- Assignment operators are used to assign values to variables.  
- Examples:  
- `=` (Assignment)  
- `+=` (Add and assign)  
- `-=` (Subtract and assign)  
- `\*=` (Multiply and assign)  
- `/=` (Divide and assign)  
- `%=` (Modulus and assign)  
  
5. \*\*Bitwise Operators\*\*:  
- Bitwise operators are used to perform operations at the bit level.  
- Examples:  
- `&` (Bitwise AND)  
- `|` (Bitwise OR)  
- `^` (Bitwise XOR)  
- `~` (Bitwise NOT)  
- `<<` (Left shift)  
- `>>` (Right shift)  
  
6. \*\*Membership Operators\*\*:  
- Membership operators are used to test if a value is a member of a sequence, like a list or a string.  
- Examples:  
- `in` (True if a value is found in the sequence)  
- `not in` (True if a value is not found in the sequence)  
  
7. \*\*Identity Operators\*\*:  
- Identity operators are used to compare the memory locations of two objects.  
- Examples:  
- `is` (True if two variables reference the same object)  
- `is not` (True if two variables reference different objects)  
  
Operators play a crucial role in controlling the flow of a program, performing calculations, making decisions, and manipulating data. Understanding how to use operators is fundamental to writing effective and efficient Python code. **Using Operators**

 USING OPERATORS  
  
Expressions can be broken up into two parts:  
  
**Operators**  
  
Operators provide the functionality to an expression and can be represented by symbols such as + or by keywords such as and. Operators require data to operate, and this data is known as an operand. In the example: 2 and 3 functions as operands for the + operator, and a and b function as the operands for the / (divide) operator.  
  
Example 1: Basic operators:  
  
>>> #Interpreter  
>>> 5 + 2  
7  
>>> 5 \* 2  
10  
>>> 5 - 2  
3  
>>>  
  
In the following list of operators, each operator has a specific purpose, and knowing how to use them is essential when writing efficient programs. These operators return a single value from their mathematical calculation:  
  
Table 1 – Mathematical operators  
  
Operation Description  
a + b Sum of a and b  
a - b Difference between a and b  
a \* b Product of a and b  
a / b Quotient of a and b  
a // b  
Floored quotient of a and b. Returns an integer value that was implicitly converted from a floating point value.  
For example, 8 / 5 = 1.6, when implicitly converted all digits fall away after the dot (.), 8 // 5 = 1  
  
a % b Remainder of a / b (modular operator)  
-a a negated  
+a a unchanged  
These functions also return a single value from their mathematical calculation:  
  
Table 2: Mathematical functions  
  
Operation Description  
abs(a) Absolute value or magnitude of a  
int(a) a converted to integer  
float(a) a converted to floating point  
complex(re, imag) A complex number with real part re, imaginary part imag. imag defaults to zero if not specified.  
divmod(a, b) The pair (a // b, a % b)  
pow(a, b) a to the power b  
a \*\* b a to the power b  
The following examples will clarify the use of operators:  
  
>>> #Interpreter  
  
>>>a = 8  
>>> b = 3  
>>> a + b #Sum of a and b  
11 >>> a - b #Difference of a and b  
5  
>>> a \* b #Product of a and b  
24  
>>> a / b #Quotient of a and b  
2.6666666666666665  
>>> a // b #Floored quotient of a and b  
2  
>>> a % b #Remainder of a / b  
2  
>>> -a  
#a negated -8  
>>> +a #a unchanged  
8 >>> abs(a) #Absolute value or magnitude of a  
8 >>> int("776") #a string converted to integer  
776  
>>> float("5.4") #a string converted to floating point  
5.4  
>>> complex(a, b) #complex number  
(8+3j)  
>>> divmod(8, 5) #The pair (a // b, a % b)  
(1, 3)  
>>> pow(a, b) #a to the power b  
512  
>>> a \*\* b #a to the power b  
512  
>>>  
  
The following functions can be used with floating point numbers and integers (float and int):  
  
Table 3: Other mathematical functions  
  
Functions Description  
math.trunc(a) a truncated to Integral  
round(a a rounded to the nearest integer  
math.floor(a) The greatest integral float <= a  
math.ceil(a) The least integral float  
NOTE Floats can use all of int’s operators and functions. ints can use float’s operators and functions, but in some cases it will not make sense to use them with ints because some functions would just return the value they were passed. For example, math.trunc(a) would return a. Some operators can also be used with other data types, e.g. str.  
Example 3: Mathematical functions:  
  
>>> >>> #Interpreter  
>>> import math #the import statement is covered in unit 2  
>>> a = 8.5  
>>> b = 5.1  
>>> math.trunc(a) #a truncated to integral (whole number)  
8  
>>> round(a) # a rounded to the nearest integer  
9  
>>> math.floor(b) #The greatest integral float <= a  
5  
>>> math.ceil(a) #The least integral float >= a  
9  
>>>  
  
There are eight comparison operations in Python. They all have the same priority, but their priority is higher than the Boolean operations (and-or-not operators). Comparisons can be chained arbitrarily, x < y <= z is equivalent to x < y and y <= z, except that y is evaluated only once (but in both cases z is not evaluated at all when x < y is found to be false).  
  
The following table summarises the comparison operators:  
  
Table 4: Comparison operators  
  
Operation Description  
< Less than  
<= Less than or equal  
> Greater than  
>= Greater than or equal  
== Equal  
!= Not equal  
is Object identity  
is not Negated object identity  
The following example shows how two numbers are compared to each other in order to find out their similarities and differences. All of these comparison operators return a Boolean True or False:  
  
>>> #Interpreter  
>>> a = 7  
>>> b = 8.8  
>>> a < b  
True >>> a <= b  
True  
>>> a > b  
False  
>>> a >= b  
False  
>>> a == b  
False  
>>> a != b  
True  
>>> a is b  
False  
>>> a is not b  
True  
>>>  
  
Some other assignment operators that have not yet been mentioned are:  
  
Operation Description  
-= The variable to the left of the operator is assigned to the variable’s value minus the value right of the assignment operator.  
\*= The variable to the left of the operator is assigned to the variable’s value multiplied by the value right of the assignment operator  
/= The variable to the left of the operator is assigned to the variable’s value divided by the value right of the assignment operator.  
%= The variable to the left of the operator is assigned to the variable’s value modulated by the value right of the assignment operator, which returns the remainder.  
Below is a short example that is self-explanatory:  
  
Example 5: Assignment operators:  
  
>>> #Interpreter  
>>> x = 16  
>>> x -= 2  
>>> x  
14  
>>> x \*= 2  
>>> x  
28  
>>> x /= 4  
>>> x  
7  
>>> x %= 3  
>>> x  
1  
>>>  
  
NOTE Just typing x is a shorter way of typing print(x). This notation can only be used while working in the interpreter, and not in the text editor.

**Activity 1**

 # Get user input for the number of liters  
liters = float(input("Enter the number of liters: "))  
  
# Calculate the number of bottles  
bottles = int(liters \* 1000 / 500) # 500 ml per bottle  
  
# Calculate the remaining water in liters  
remaining\_liters = round(liters - (bottles \* 0.5), 2)  
  
# Print the result  
print(f"{liters:.2f}L water will fill {bottles} bottles ({remaining\_liters}L remains)")

**Activity 2**

 # Constants  
MEGALITRE\_TO\_LITRE = 1000000 # 1 megalitre = 1,000,000 litres  
DAM\_CAPACITY\_MEGALITRE = 1  
  
# Get user input for millilitres of water released  
millilitres\_released = int(input("Enter the millilitres of water released: "))  
  
# Convert millilitres to litres  
litres\_released = millilitres\_released / 1000 # 1,000 millilitres = 1 litre  
  
# Calculate litres remaining in the dam  
litres\_remaining = DAM\_CAPACITY\_MEGALITRE \* MEGALITRE\_TO\_LITRE - litres\_released  
  
# Calculate the percentage of litres remaining  
percentage\_remaining = (litres\_remaining / (DAM\_CAPACITY\_MEGALITRE \* MEGALITRE\_TO\_LITRE)) \* 100  
  
# Generate the report  
print(f"Millilitres released: {millilitres\_released}")  
print(f"Litres released: {litres\_released}")  
print(f"Litres remaining: {litres\_remaining}")  
print(f"Percentage of litres remaining: {percentage\_remaining:.2f}%")