

Introduction to Python

Day 1

Python – An Introduction

Why Python ?

- Its ease of use - For those who are new to coding and programming, Python can be an excellent first step. It's relatively easy to learn.
- Its simple syntax - Python is relatively easy to read and understand, as its syntax is more like English
- Its thriving community- As it's an open-source language, anyone can use Python to code. What's more, there is a community that supports and develops the ecosystem, adding their own contributions and libraries
- Its versatility - As we'll explore in more detail, there are many uses for Python. Whether you're interested in data visualization, artificial intelligence or web development, you can find a use for the language.

What Python can be used for? – Pretty much everything!

Use cases

- AI and machine learning - Python is favorite languages among data scientists, and there are many Python machine learning and AI libraries and packages available
- Data analytics - Python for data science and analytics makes sense. The language is easy-to-learn, flexible, and well-supported, meaning it's relatively quick and easy to use for analyzing data
- Data visualisation - Python provides a variety of graphing libraries with all kinds of features. Whether you're looking to create a simple graphical representation or a more interactive plot, you can find a library to match your needs – Seaborn, Matplotlib, Plotly etc
- Programming applications - The general-purpose language can be used to read and create file directories, create GUIs and APIs, and more. Whether it's blockchain applications, audio and video apps, or machine learning applications, you can build them all with Python

What Python can be used for? – Pretty much everything!

Use cases

- Web development - Python is a great choice for web development. This is largely due to the fact that there are many Python web development frameworks to choose from, such as Django, Pyramid, and Flask. These frameworks have been used to create sites and services such as Spotify, Reddit and Mozilla
- Game Development - It's possible to create simple games using the programming language, which means it can be a useful tool for quickly developing a prototype.
- Language development- The simple and elegant design of Python and its syntax means that it has inspired the creation of new programming languages. Languages such as Cobra, Coffee Script, and Go all use a similar syntax to Python.
- Web scrapping
- CAD Application
- Embedded System Development

Python – an interpreted/compiled language?

To answer this ,let's first understand what compiled and interpreted language means?

A typically compiled language consists of following steps-

- Pre-processing
- Compilation
- Assembling
- Linking

Python – an interpreted/compiled language?

Pre-processing

It is done by preprocessor which takes the source code written by the user and processes it – it includes removing source code, expanding the methods/functions used in the source code and including the files used in header of the code. For example, In C, when we include a header file using the `#include<filename>` statement, the preprocessor consists of the file in the source code. Similarly, when defining constants and expressions using the `#define` statements or macros, the preprocessor expands the symbolic constants at each occurrence. It replaces them using the value or expression defined in the `#define` statement.

Python – an interpreted/compiled language?

Compilation

- Once preprocessing is done, the compiler receives the modified source code and converts it to assembly language. This process is called compilation.
- Assembly language is an intermediate language between machine code and high-level language and can be understood by humans. It is highly optimized, and all the instructions, including memory locations and registers which are to be used in the program, are specified in this step

Python – an interpreted/compiled language?

Assembling

- After compilation, the program is converted into assembly language code. It is then processed by an assembler and is converted into machine-readable binary code. **The process of converting an assembly language code into machine language is called assembling**

Linking

- It is the last and final step in the process of compilation. A linker performs linking. The linker collects all the machine codes from different modules to be executed and merges them into a single object file.
- Linking is done to merge all the machine code defined in different libraries so that the program written by the user runs successfully

Python – an interpreted/compiled language?

What is an interpreter?

An interpreter converts the code written in a high-level language into an efficient intermediate code called bytecode. The bytecode is then executed line by line on a virtual machine to produce the output

Python – an interpreted/compiled language?

What is an interpreted language?

- An interpreted language is a programming language that an interpreter executes to produce outputs.
- Interpreted languages are platform-independent; this is due to the fact that they are run in a virtual machine and need not be specific about the hardware of the machine in which they are being executed.

Python – an interpreted/compiled language?

It is somewhere in between or both of them – in parts!

Python – an interpreted/compiled language?

- When we try to execute a python program, the interpreter processed the source code and is converted into bytecode. Bytecodes are low-level codes understandable by the virtual machine. The bytecode is then executed line by line by the virtual machine
- As we know that converting a source code from one language to another is termed compilation, We can say that a python program is also compiled. But to create a bytecode and not a machine code. We can also say that the python program is first compiled to create the bytecode, and then the bytecode is executed line by line by the virtual machine.

Python – an interpreted/compiled language?

- Python can be exclusively defined as a compiled language or an interpreted language. This is because, during the execution of a python program, it is first compiled to create the bytecode. Then the virtual machine interprets bytecode to line by line to produce the outputs

What are Data Types and Variables in Python

The **data type** of an item defines the type and range of values that item can have.

Python Data Types

Unlike many other languages, Python does not place a strong emphasis on defining the data type of an object, which makes coding much simpler. The language provides three main data types:

- **Numbers**
- **Strings**
- **Booleans**

What are Data Types and Variables in Python

The **data type** of an item defines the type and range of values that item can have.

Python Data Types

Unlike many other languages, Python does not place a strong emphasis on defining the data type of an object, which makes coding much simpler. The language provides three main data types:

- **Numbers**
- **Strings**
- **Booleans**

What are Data Types and Variables in Python

A **variable** is simply a name to which a value can be *assigned*.

- Variables allow us to give meaningful names to data.
- A big advantage of variables is that they allow us to store data so that we can use it later to perform operations in the code.
- Variables are mutable. Hence, the value of a variable can always be updated or replaced.
- The simplest way to assign a value to a variable is through the = operator.



Naming Convention for Variables

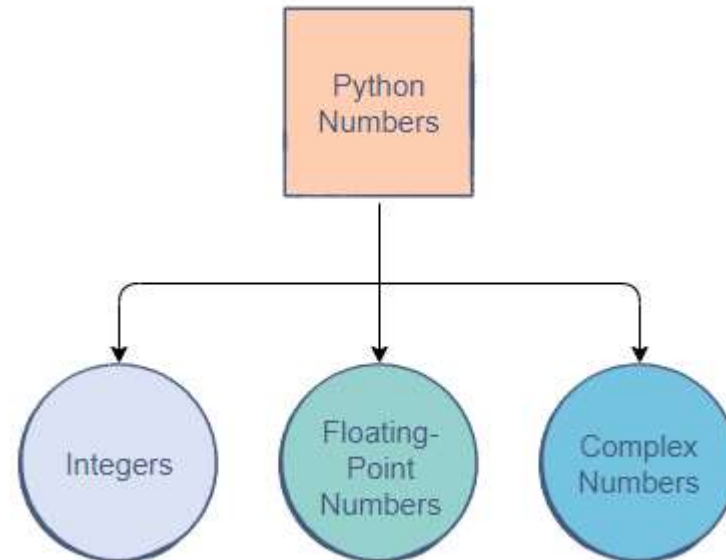
- The name can start with an upper or lower case alphabet.
 - All the names are case sensitive.
 - A number can appear in the name, but not at the beginning.
 - The `_` character can appear anywhere in the name.
 - Spaces are not allowed. Instead, we must use `snake_case` to make variable names readable.
 - The name of the variable should be something meaningful that describes the value it holds, instead of being random characters.
- For example, `inc` or even `income` would not give any useful information but names like `weekly_income`, `monthly_income`, or `annual_income` explain the purpose of our defined variable

Naming Convention for Variables

- The name can start with an upper or lower case alphabet.
 - All the names are case sensitive.
 - A number can appear in the name, but not at the beginning.
 - The `_` character can appear anywhere in the name.
 - Spaces are not allowed. Instead, we must use `snake_case` to make variable names readable.
 - The name of the variable should be something meaningful that describes the value it holds, instead of being random characters.
- For example, `inc` or even `income` would not give any useful information but names like `weekly_income`, `monthly_income`, or `annual_income` explain the purpose of our defined variable

Numbers

There are three main types of numbers in Python



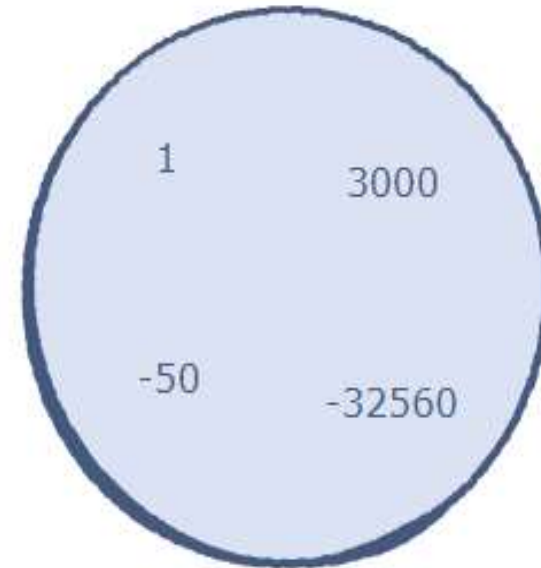
Integers

In Python, all negative numbers start with the - symbol.

Integers

The integer data type is comprised of all the positive and negative whole numbers.

The amount of memory an integer occupies depends on its value. For example, `0` will take up 24 *bytes* whereas `1` would occupy 28 *bytes*.



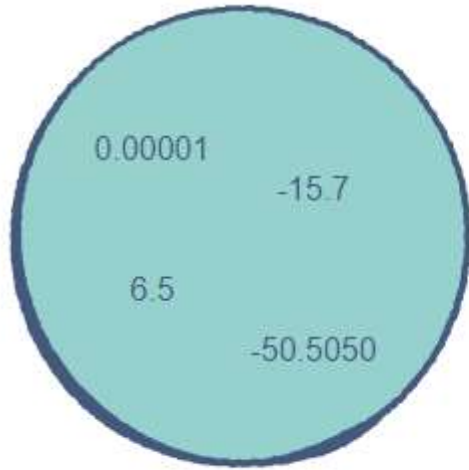
Integers

So, what's the size of integer that you can store in Python 3.0+? – Nearly unlimited sized integer no can be stored.

```
print(10) # A positive integer  
print(-3000) # A negative integer
```

```
num = 123456789 # Assigning an integer to a variable  
print(num)  
num = -16000 # Assigning a new integer  
print(num)
```

Floats



Floating-point numbers in Python.

Floating Point Numbers

Floating-point numbers, or **floats**, refer to positive and negative decimal numbers.

Python allows us to create decimals up to a very high decimal place.

This ensures accurate computations for precise values.

A float occupies *24 bytes* of memory.

Floats

```
print(1.000000000005) # A positive float
```

```
print(-85.6701) # A negative float
```

```
flt_pt = 1.23456789
```

```
print(flt_pt)
```

Complex numbers

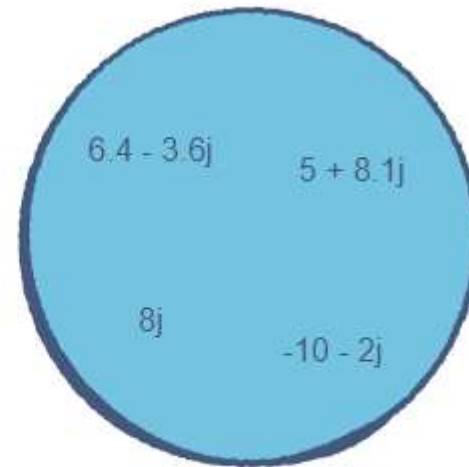
Complex numbers are useful for modelling physics and electrical engineering models in Python. While they may not seem very relevant right now, it never hurts to know

Complex Numbers

Python also supports complex numbers, or numbers made up of a real and an imaginary part.

Just like the `print()` statement is used to print values, `complex()` is used to create complex numbers.

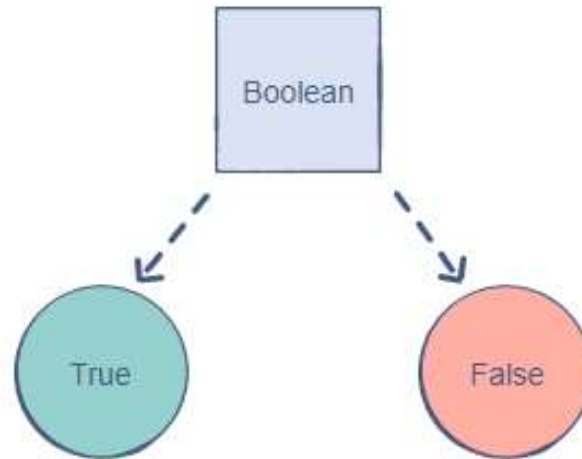
It requires two values. The first one will be the *real* part of the complex number, while the second value will be the *imaginary* part.



Complex numbers in Python.

Boolean

The Boolean (also known as bool) data type allows us to choose between two values: True or False.



Boolean

A Boolean is used to determine whether the logic of an expression or a comparison is correct. It plays a huge role in data comparisons.

```
print(True)
```

```
f_bool = False  
print(f_bool)
```

String

A string is a collection of characters closed within single, double or triple quotation marks. In Python 3.0, strings are Unicode strings

A string can also contain a single character or be entirely empty.

```
print("Harry Potter!") # Double quotation marks
```

```
got = 'Game of Thrones...' # Single quotation marks  
print(got)  
print("$") # Single character
```

```
empty = ""  
print(empty) # Just prints an empty line
```

```
multiple_lines = """Triple quotes allows  
multi-line string."""  
print(multiple_lines)
```

String Immutability

Once we assign a value to a string, we can't update it later

```
string = "Immutability"  
string[0] = 'O' # Will give error
```

The above code gives `TypeError` because Python doesn't support item assignment in case of strings.

String Immutability

```
str1 = "hello"  
print(id(str1))
```

```
str1 = "bye"  
print(id(str1))
```

Check out for the Ids (memory locations where the String variables are actually stored).

Logical Operators

The operators which act on one or two boolean values and return another boolean value are called logical operators. There are 3 key logical operators. Let X and Y be the two operands and let **X = True** and **Y = False**.

Operator	Description	Example
and	<u>Logical AND</u> : If both the operands are true then the condition is true.	(X and Y) is false
or	<u>Logical OR</u> : If any of the two operands are then the condition is true.	(X or Y) is true
not	<u>Logical NOT</u> : Used to reverse the logical state of its operand.	Not(X) is false

Truth Table

The Truth table for all combination of values of X and Y

X	Y	X and Y	X or Y	not(X)	not(Y)
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Assignment – Relational Operators

```
x = 9
y = 13

print('x > y is',x > y) # Here 9 is not greater than 13

print('x < y is',x < y) # Here 9 is Less than 13

print('x == y is',x == y) # Here 9 is not equal to 13

print('x != y is',x != y) # Here 9 is not equal to 13

print('x >= y is',x >= y) # Here 9 is not greater than or equal to 13

print('x <= y is',x <= y) # Here 9 is Less than 13
```


Solution

```
x > y is False  
x < y is True  
x == y is False  
x != y is True  
x >= y is False  
x <= y is True
```

Assignment – Logical Operators

```
x = True
y = False

print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

Solution

```
x and y is False  
x or y is True  
not x is False
```

Comments in Python

Comments are descriptions that help programmers better understand the intent and functionality of the program. They are completely ignored by the Python interpreter.

In Python, we use the hash symbol `#` to write a single-line comment. For example,

```
# Program to print "Hello World"  
print("Hello World")
```

Python Comments Using Strings

If we do not assign strings to any variable, they act as comments. For example,

```
"I am a single-line comment"  
'''
```

```
I am a  
multi-line comment!  
'''
```

```
print("Hello World")
```

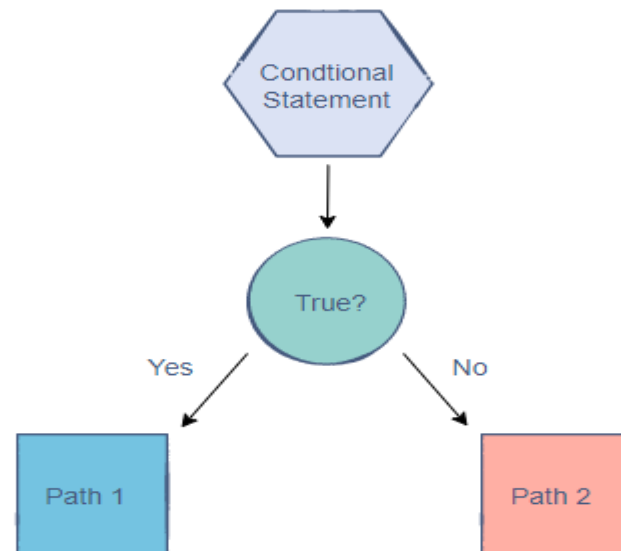
What are Conditional Statements?

A conditional statement is a Boolean expression that, if True, executes a piece of code.

It allows programs to branch out into different paths based on Boolean expressions result in True or False outcomes.

In this way, conditional statements control the flow of the code and allow the computer to think. Hence, they are classified as control structures.

Conditional statements are an integral part of programming that every coder needs to know



Conditional Statements in Python

To handle conditional statements, Python follows a particular convention:

if conditional statement is True:

 # execute expression1

 pass

else:

 # execute expression2

 pass

There are three types of conditional statements in Python:

➤ if

➤ if-else

➤ if-elif-else

The If Structure

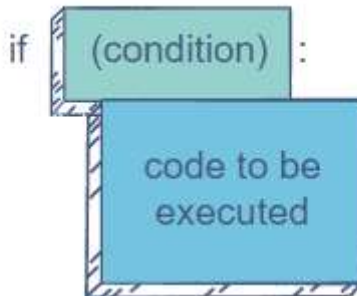
The if statement is the foundation of conditional programming in Python.

The simplest conditional statement that we can write is the **if statement (it is not loop)**. It comprises of two parts:

The condition

The code to be executed

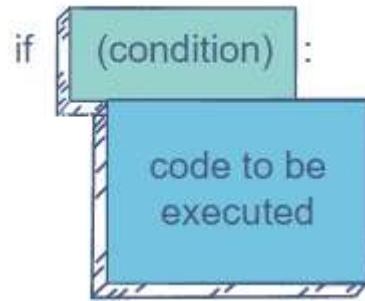
The : in the illustration above is necessary to specify the beginning of the if statement's code to be executed. However, the parentheses, (), around the condition are optional. The code to be executed is indented at least one tab to the right.



The Structure

Indentation

Indentation plays an essential role in Python. Statements with the same level of indentation belong to the same block of code. The code of an if statement is indented a space further than the code outside it in order to indicate that this is an inner and inter-related block



The Flow of an if Statement

if the condition holds True, execute the code to be executed. Otherwise, skip it and move on.

```
num = 5
if (num == 5): # The condition is true
    print("The number is equal to 5") # The code is executed
if num > 5: # The condition is false
    print("The number is greater than 5") # The code is not executed
```

Our first condition simply checks whether the value of num is 5. Since this Boolean expression returns True, the compiler goes ahead and executes the print statement on line 4.

As we can see, the print command inside the body of the if statement is indented to the right. If it wasn't, there would be an error. Python puts a lot of emphasis on proper indentation

Conditions with Logical Operators

We can use logical operators to create more complex conditions in the if statement. For example, we may want to satisfy multiple clauses for the expression to be True.

```
num = 12
if num % 2 == 0 and num % 3 == 0 and num % 4 == 0:
    # Only works when num is a multiple of 2, 3, and 4
    print("The number is a multiple of 2, 3, and 4")
if (num % 5 == 0 or num % 6 == 0):
    # Only works when num is either a multiple of 5 or 6
    print("The number is a multiple of 5 and/or 6")
```

In the first if statement, all the conditions have to be fulfilled since we're using the and operator.

In the second if statement, the Boolean expression would be true if either or both of the clauses are satisfied because we are using the or operator.

The Flow of an if Statement

if the condition holds True, execute the code to be executed.
Otherwise, skip it and move on.

```
num = 5
if (num == 5): # The condition is true
    print("The number is equal to 5") # The code is executed
if num > 5: # The condition is false
    print("The number is greater than 5") # The code is not executed
```

Our first condition simply checks whether the value of num is 5. Since this Boolean expression returns True, the compiler goes ahead and executes the print statement on line 4.

As we can see, the print command inside the body of the if statement is indented to the right. If it wasn't, there would be an error. Python puts a lot of emphasis on proper indentation

Nested if Statements

A great feature of conditional statements is that we can nest them. This means that there could be an if statement inside another! Variables allow us to give meaningful names to data.

```
num = 63
```

```
if num >= 0 and num <= 100:
```

```
    if num >= 50 and num <= 75:
```

```
        if num >= 60 and num <= 70:
```

```
            print("The number is in the 60-70 range")
```

Note: Each nest if statement requires further indentation.

Creating and Editing Values

In a conditional statement, we can edit the values of our variables.

Furthermore, we can create new variables.

```
num = 10
if num > 5:
    num = 20 # Assigning a new value to num
    new_num = num * 5 # Creating a new value called newNum

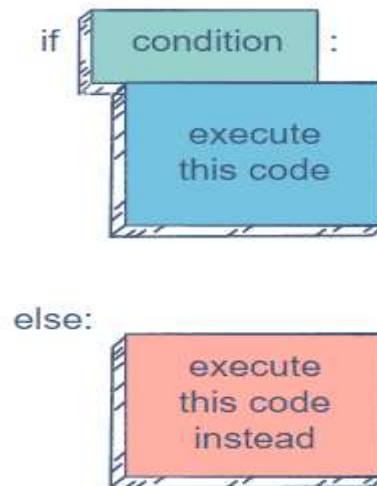
# The if condition ends, but the changes made inside it remain
print(num)
print(new_num)
```

The if-else Statement

What if we want to execute a different set of operations in case an if condition turns out to be False?

That is where the if-else statement comes into the picture.

The if-else statement looks something like this:

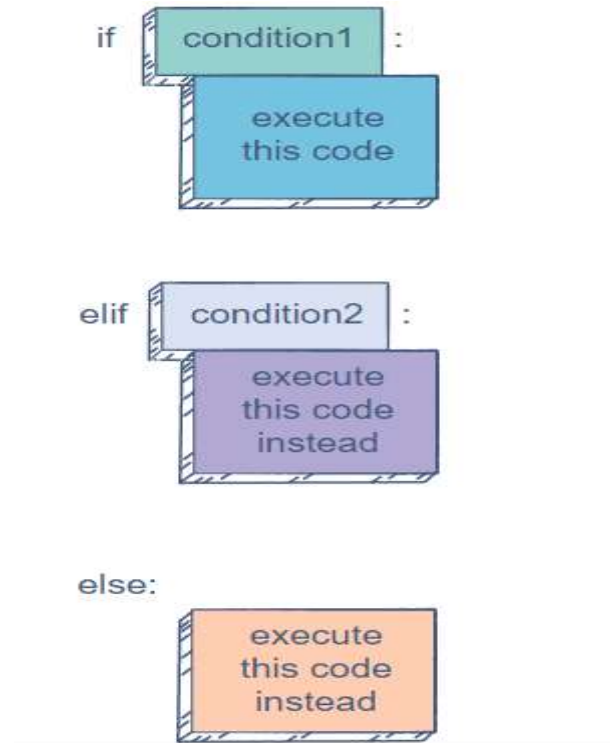


The if-elif-else Statement

The if-else statement handles two sides of the same condition: True and False. This works very well if we're working with a problem that only has two outcomes.

But, if multiple outcomes are possible, the if-elif-else statement shines. It is the most comprehensive conditional statement because it allows us to create multiple conditions easily.

The elif stands for else if, indicating that if the previous condition fails, try this one.



The if-elif-else Statement

```
light = "Red"
if light == "Green":
    print("Go")
elif light == "Yellow":
    print("Caution")
elif light == "Red":
    print("Stop")
else:
    print("Incorrect light signal")
```

Now, our conditional statement caters to all possible values of light. Please run in PyCharm Debugger Mode. Change values and see the execution

Multiple elif Statements

This is the beauty of the if-elif-else statement. We can have as many elifs as we require, as long as they come between if and else

Note: An if-elif statement can exist on its own without an else block at the end. However, an elif cannot exist without an if statement preceding it (which naturally makes sense).



Multiple elif statements...

Diff between multiple ifs and if-elif-else

if-elif-else or if-elif statement is not the same as multiple if statements. if statements act independently.

If the conditions of two successive ifs are True, both statements will be executed.

On the other hand, in if-elif-else, when a condition evaluates to True, the rest of the statement's conditions are not evaluated.

```
num = 10
if num > 5:
    print("The number is greater than 5")
if num % 2 == 0:
    print("The number is even")
if not num % 2 == 0:
    print("The number is odd")
```

As we can see, in the if tab, all the statements are computed one by one. Hence, we get multiple outputs.

Exercise

You must discount a price according to its value.

If the price is 300 or above, there will be a 30% discount.

If the price is between 200 and 300 (200 inclusive), there will be a 20% discount.

If the price is between 100 and 200 (100 inclusive), there will be a 10% discount.

If the price is less than 100, there will be a 5% discount.

If the price is negative, there will be no discount.

Exercise Solution

```
price = 250
if price >= 300:
    price *= 0.7 # (1 - 0.3)
elif price >= 200:
    price *= 0.8 # (1 - 0.2)
elif price >= 100:
    price *= 0.9 # (1 - 0.1)
elif price < 100 and price >= 0:
    price *= 0.95 # (1 - 0.05)
print(price)
```

To handle all the cases, we'll use an if-elif statement. Notice that we only need to specify the lower bound in each condition. This is because, for every condition, the upper bound is already being checked in the previous condition.

The program will only go into the first elif if the price is lower than 300.

This sort of smart structuring in a conditional statement can be very useful when dealing with a large number of complex cases.

Loops in Python

A loop is a control structure that is used to perform a set of instructions for a specific number of times.

Loops solve the problem of having to write the same set of instructions over and over again. We can specify the number of times we want the code to execute.

One of the biggest applications of loops is traversing data structures, e.g. lists, tuples, sets, etc. In such a case, the loop iterates over the elements of the data structure while performing a set of operations each time.

Just like conditional statements, a loop is classified as a control structure because it directs the flow of a program by making varying decisions in its iterations. Loops are a crucial part of many popular programming languages such as C++, Java, and JavaScript too.

Loops in Python

There are two types of loops that we can use in Python:

The **for** loop

The **while** loop

Both differ slightly in terms of functionality

The for Loop

- A for loop uses an iterator to traverse a sequence, e.g. a range of numbers, the elements of a list, etc. In simple terms, the iterator is a variable that goes through the list.
- The iterator starts from the beginning of the sequence. In each iteration, the iterator updates to the next value in the sequence.
- The loop ends when the iterator reaches the end.

The for Loop

Structure

In a for loop, we need to define three main things:

- The name of the iterator
- The sequence to be traversed
- The set of operations to perform

The loop always begins with the for keyword. The body of the loop is indented to the right:

```
for iterator in sequence :
```



The `in` keyword specifies that the iterator will go through the values *in* the sequence/data structure.

The for Loop

Looping Through a Range

In Python, the built-in `range()` function can be used to create a sequence of integers. This sequence can be iterated over through a loop. A range is specified in the following format:

```
range(start, end, step)
```

The end value is not included in the list.

If the start index is not specified, its default value is 0.

The step decides the number of steps the iterator jumps ahead after each iteration. It is optional and if we don't specify it, the default step is 1, which means that the iterator will move forward by one step after each iteration.

The for Loop

Example

```
for i in range(1, 11): # A sequence from 1 to 10
    if i % 2 == 0:
        print(i, " is even")
    else:
        print(i, " is odd")
```

As we can see above, rather than individually checking whether every integer from 1 to 10 is even or odd, we can loop through the sequence and compute $i \% 2 == 0$ for each element.

The iterator, i , begins from 1 and becomes every succeeding value in the sequence.

The for Loop

If you want to change the **step** in for loop-

```
for i in range(1, 11, 3): # A sequence from 1 to 10 with a step of 3
    print(i)
```

Looping Through a List/String

A list or string can be iterated through its indices.

```
float_list = [2.5, 16.42, 10.77, 8.3, 34.21]
print(float_list)
```

```
for i in range(0, len(float_list)): # Iterator traverses to the last index of the list
    float_list[i] = float_list[i] * 2

print(float_list)
```

The for Loop

We could also traverse the elements of a list/string directly through the iterator. In the float_list above, let's check how many elements are greater than 10:

```
float_list = [2.5, 16.42, 10.77, 8.3, 34.21]
```

```
count_greater = 0
```

```
for num in float_list: # Iterator traverses to the last index of the list
```

```
    if num > 10:
```

```
        count_greater += 1
```

```
print(count_greater)
```

Nested for Loops

Execution of Nested Loops

Python lets us easily create loops within loops. There's only one catch: the inner loop will always complete before the outer loop.

For each iteration of the outer loop, the iterator in the inner loop will complete its iterations for the given range, after which the outer loop can move to the next iteration.

Nested for Loops

Using a Nested for Loop

We want to print two elements whose sum is equal to a certain number n .

The simplest way would be to compare every element with the rest of the list. A nested for loop is perfect for this:

```
n = 50
```

```
num_list = [10, 4, 23, 6, 18, 27, 47]
```

```
for n1 in num_list:
```

```
    for n2 in num_list: # Now we have two iterators
```

```
        if(n1 + n2 == n):
```

```
            print(n1, n2)
```

Loops

The break Keyword

If we need to exit the loop before it reaches the end as we don't need to make any more computations in the loop.

Say, n_1 is 23 and n_2 is 27. Our condition of $n_1 + n_2 == n$ has been fulfilled but loop still runs. This is why the pair is printed twice. It would be nice to just stop it when the pair is found once.

That's what the **break** keyword is for. It can break the loop whenever we want.

```
n = 50
num_list = [10, 4, 23, 6, 18, 27, 47]
found = False # This bool will become true once a pair is found
for n1 in num_list:
    for n2 in num_list:
        if(n1 + n2 == n):
            found = True # Set found to True
            break # Break inner loop if a pair is found
if found:
    print(n1, n2) # Print the pair
    break # Break outer loop if a pair is found
```

Loops

The continue Keyword

When the continue keyword is used, the rest of that particular iteration is skipped. The loop continues on to the next iteration. We can say that it doesn't break the loop, but it skips all the code in the current iteration and moves to the next one.

```
num_list = list(range(0, 10))
```

```
for num in num_list:  
    if num == 3 or num == 6 or num == 8:  
        continue  
    print(num)
```

The loop goes into the if block when num is 3, 6, or 8. When this happens, continue is executed and the rest of the iteration, including the print() statement, is skipped.

Loops

The pass Keyword

In all practical meaning, the pass statement does nothing to the code execution. It can be used to represent an area of code that needs to be written. Hence, it is simply there to assist you when you haven't written a piece of code but still need your entire program to execute

```
num_list = list(range(20))
```

```
for num in num_list:
```

```
    pass # You can write code here later on
```

```
print(len(num_list))
```

Quiz Time:

1.What will this print?

```
string_list = ["Anakin", "Luke", "Rey", "Leia", "Vader"]  
for s in string_list:  
    if len(s) < 5:  
        print(len(s))
```

Quiz Time:

2 What error occurs when you execute the following Python code snippet?

```
apple = mango
```

- a) `SyntaxError`
- b) `NameError`
- c) `ValueError`
- d) `TypeError`

Quiz Time:

3. What will be the output of the following Python code?

```
if (9 < 0) and (0 < -9):
```

```
    print("hello")
```

```
elif (9 > 0) or False:
```

```
    print("good")
```

```
else:
```

```
    print("bad")
```

Quiz Time:

4. What will be the output of the following Python code?

```
i = 1
```

```
while True:
```

```
    if i % 3 == 0:
```

```
        break
```

```
    print(i)
```

```
    i=i+1
```

Quiz Time:

5. What will be the output of the following Python code?

```
True = False
```

```
while True:
```

```
    print(True)
```

```
    break
```

Quiz Time:

6. Output of the program?

x=3

if x>2 or x<5 and x==6:

 print("Bye")

else:

 print("Thankyou")

Quiz Time:

7. Which of these is not a core data type?

- a) Lists
- b) Dictionary
- c) Tuples
- d) Class

Print Pattern in Python using Loop

1. Decide the number of rows and columns

There is a typical structure to print any pattern, i.e., the number of rows and columns. We need to use two loops to print any pattern, i.e., use nested loops.

The outer loop tells us the number of rows, and the inner loop tells us the column needed to print the pattern.

Accept the number of rows from a user using the `input()` function to decide the size of a pattern.

2. Iterate rows

Next, write an outer loop to iterate the number of rows using a `for` loop and `range()` function.

3. Iterate columns

Next, write the inner loop or nested loop to handle the number of columns. The internal loop iteration depends on the values of the outer loop.

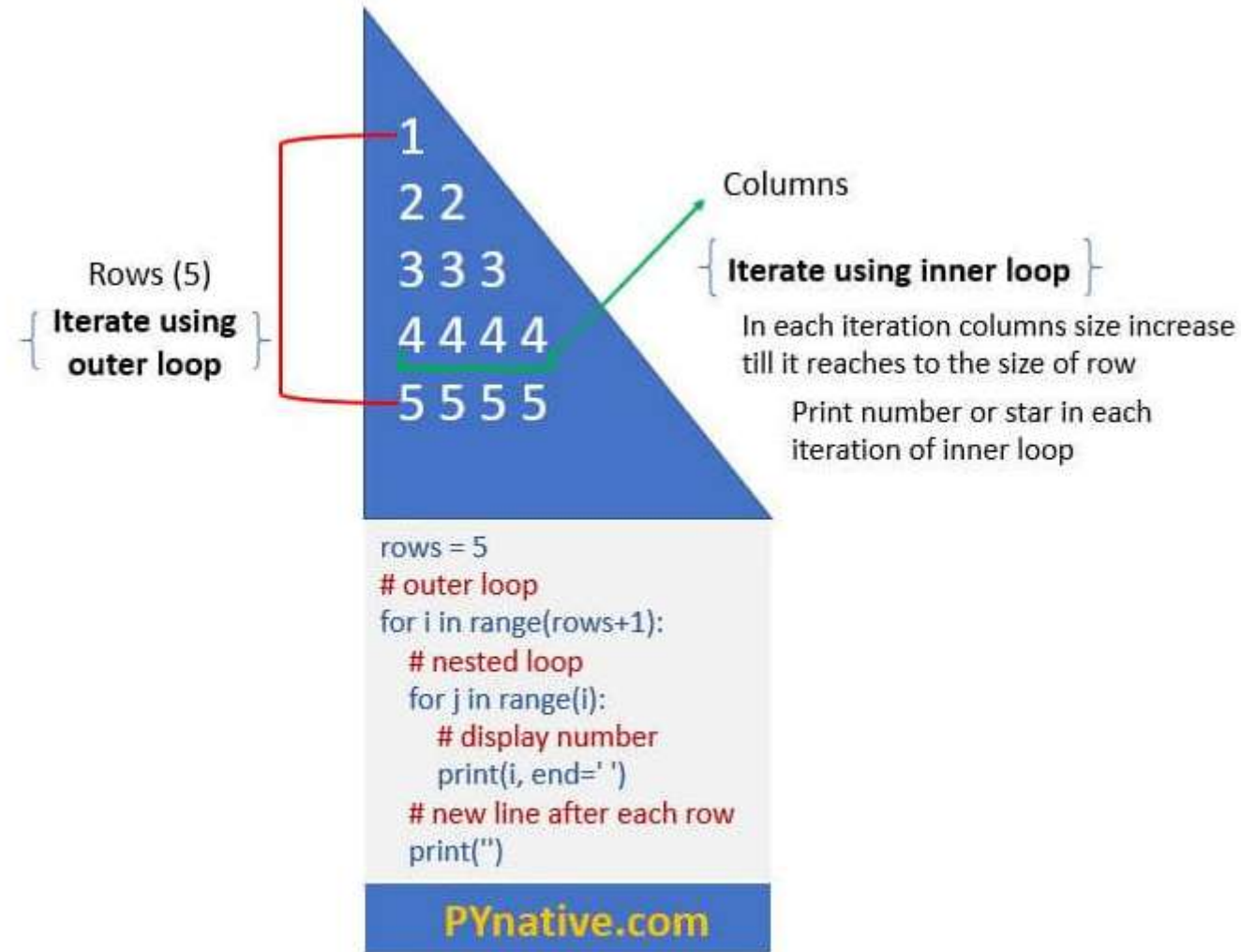
4. Print star or number

Use the `print()` function in each iteration of nested `for` loop to display the symbol or number of a pattern (like a star (asterisk `*`) or number).

5. Add new line after each iteration of outer loop

Add a new line using the `print()` function after each iteration of the outer loop so that the pattern displays appropriately.

Print Pattern in Python using Loop



The while Loop

- The while loop keeps iterating over a certain set of operations as long as a certain condition holds True.
- It operates using the following logic-

While this condition is true, keep the loop running.


The while Loop

Structure

In a for loop, the number of iterations is fixed since we know the size of the sequence.

On the other hand, a while loop is not always restricted to a fixed range. Its execution is based solely on the condition associated with it.

```
while condition is true :
```



Loop over
this set of
operations

The while Loop

Assignment

Use while loop to find out the maximum power of n before the value exceeds 1000:

```
n = 2 # Could be any number
power = 0
val = n
while val < 1000:
    power += 1
    val *= n
print(power)
```

In each iteration, we update `val` and check if its value is less than 1000. The value of `power` tells us the maximum power n can have before it becomes greater than or equal to 1000. Think of it as a counter.

The while Loop

We can also use while loops with data structures, especially in cases where the length of data structure changes during iterations.

The following loop computes the sum of the first and the last digits of any integer:

```
n = 249
```

```
last = n % 10 # Finding the last number is easy
```

```
first = n # Set it to `n` initially
```

```
while first >= 10:
```

```
    first //= 10 # Keep dividing by 10 until the leftmost digit is reached.
```

```
result = first + last
```

```
print(result)
```

Caution while using while loop

Compared to for loops, we should be more careful when creating while loops. This is because a while loop has the potential to never end. This could crash a program!

Have a look at these simple loops:

```
while(True):  
    print("Hello World")
```

```
x = 1  
while(x > 0):  
    x += 5
```

The break, continue, and pass keywords work with while loops.

Like for loops, we can also nest while loops. Furthermore, we can nest the two types of loops with each other.

Iteration vs. Recursion

- If we observe closely, there are several similarities between iteration and recursion. In recursion, a function performs the same set of operations repeatedly but with different arguments.
- A loop does the same thing except that the value of the iterator and other variables in the loop's body change in each iteration.
- Figuring out which approach to use is an intuitive process. Many problems can be solved through both.
- Recursion is useful when we need to divide data into different chunks. Iteration is useful for traversing data and also when we don't want the program's scope to change.

Assignment

1. Find the sum of first **n** natural numbers using both **while** and **loop**

2. Given a number **n**, check whether the number is a **prime** number or not.

Hint: A prime number is always positive so we are checking that at the beginning of the program. Next, we are dividing the input number by all the numbers in

the range of 2 to (number - 1) to see whether there are any positive divisors other than 1 and the number itself (Condition for Primality). If any divisor is found then we display, "Is Prime", else we display, "Is Not Prime".

3. Given an Integer, Print all the Prime Numbers between 0 and that Integer.

4. Given a list, write `check_sum()` function which takes in the list and returns True if the sum of two numbers in the list is zero. If no such pair exists, return False.

5. the Fibonacci sequence is a series of numbers where every number is the sum of the two numbers before it. The first two numbers are 0 and 1. Use loop to print nth Fibonacci number.

Assignment

1. Find the sum of first **n** natural numbers using both **while** and **loop**

2. Given a number **n**, check whether the number is a **prime** number or not.

Hint: A prime number is always positive so we are checking that at the beginning of the program. Next, we are dividing the input number by all the numbers in

the range of 2 to (number - 1) to see whether there are any positive divisors other than 1 and the number itself (Condition for Primality). If any divisor is found then we display, "Is Prime", else we display, "Is Not Prime".

3. Given an Integer, Print all the Prime Numbers between 0 and that Integer.

4. Given a list, write `check_sum()` function which takes in the list and returns True if the sum of two numbers in the list is zero. If no such pair exists, return False.

5. the Fibonacci sequence is a series of numbers where every number is the sum of the two numbers before it. The first two numbers are 0 and 1. Use loop to print nth Fibonacci number.

Solution 1

n = 4

sum = 0

i = 1 #Initialising the looping variable to 1

while (i<=n): #The loop will continue till the value of i<number

 sum = sum + i

 i = i+1 #Value of i is updated at the end of every iteration

print(sum)

Solution 2

```
number=15
isPrime= True #Boolean to store if number is prime or not
if number > 1: # prime number is always greater than 1
    i=2
    while i< number:
        if (number % i) == 0: # Checking for positive divisors
            isPrime= False
            break
        i=i+1
if(number<=1): # If number is less than or equal to 1
    print("Is Not Prime")
elif(isPrime): # If Boolean is true
    print("Is Prime")
else:
    print("Is Not Prime")
```

Solution 3

```
n= 15 #assigning n
```

```
k=2 # Looping variable starting from 2
```

```
while k<=n:# Loop will check all numbers till n
```

```
    d=2 # The inner loop also checks all numbers starting from 2
```

```
    isPrime = False
```

```
    while d<k:
```

```
        if(k%d==0):
```

```
            isPrime = True
```

```
            d=d+1
```

```
    if(not(isPrime)):
```

```
        print(k)
```

```
    k=k+1
```

Solution 4

```
num_list = [10, -14, 26, 5, -3, 13, -5]
```

```
def check_sum(num_list):
```

```
    for first_num in range(len(num_list)):
```

```
        for second_num in range(first_num, len(num_list)):
```

```
            if num_list[first_num] + num_list[second_num] == 0:
```

```
                return True
```

```
    return False
```

```
print(check_sum(num_list))
```

Solution 5

n=7

```
def fib(n):
```

```
    first = 0
```

```
    second = 1
```

```
    if n < 1:
```

```
        return -1
```

```
    if n == 1:
```

```
        return first
```

```
    if n == 2:
```

```
        return second
```

```
    count = 3
```

```
    while count <= n:
```

```
        fib_n = first + second
```

```
        first = second
```

```
        second = fib_n
```

```
        count += 1
```

```
    return fib_n
```

Functions in Python

Remember the `print()` statement? It always performs predefined task. Well, it turns out that it was function all along!

Why Use Functions:

Functions are useful because they make the code concise and simple. The primary benefits of using functions are:

Reusability: Function be used over and over again For example, a `sum()` function could compute the sum of all the integers we provide it. We won't have to write the summing operation ourselves each time.

Neat code:A code containing functions is concise and easy to read.

Modularization: : Functions help in modularizing code. Modularization means dividing the code into smaller modules, each performing a specific task.

Easy Debugging-: It is easy to find and correct the error in a function as compared to raw code

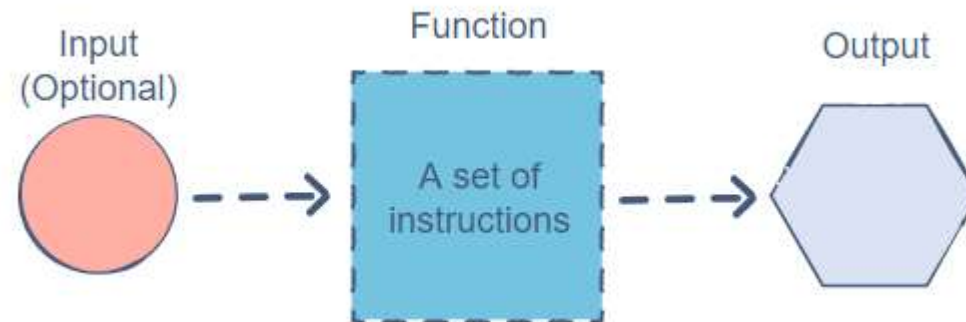
Type of Functions in Python

We can divide functions into the following two types:

- User-defined functions: Functions that are defined by the users. Eg. The `add()` function that we can create to add 2 numbers. Similarly `check_sum()` function that we created earlier. These functions are also called as custom functions.
- Inbuilt Functions: Functions that are inbuilt in python. Eg. The `print()` function.

Functions in Python

In other words ,this is what function is



Functions in Python

Suppose we want to find the smaller value between two integers: Suppose we want to find the smaller value between two integers:

```
num2 = 40
if num1 < num2:
    minimum = num1
else:
    minimum = num2
print(minimum)
```

```
num1 = 250
num2 = 120
if num1 < num2:
    minimum = num1
else:
    minimum = num2
print(minimum)
```

Function in Python

What if we use a built-in function `min(number1,number2)` instead of writing comparison logic over and over again-

```
minimum = min(10, 40)  
print(minimum)
```

```
minimum = min(10, 100, 1, 1000) # It even works with multiple arguments  
print(minimum)
```

```
minimum = min("Superman", "Batman") # And with different data types  
print(minimum)
```

Function in Python

Structure

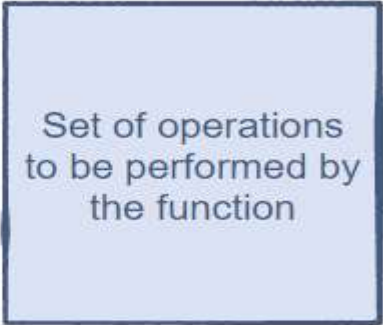
In Python, a function can be defined using the `def` keyword in the following format:

The function name is simply the name we'll use to identify the function.

The parameters of a function are the inputs for that function. We can use these inputs within the function. Parameters are optional. We'll get to know more about these later.

The body of the function contains the set of operations that the function will perform. This is always indented to the right.

```
def function name (parameters) :
```



Set of operations
to be performed by
the function

Function in Python

Example

Let's make a plain function that prints four lines of text. It won't have any parameters. We'll name it `my_print_function`.

We can call the function in our code using its name along with empty parentheses:

```
def my_print_function(): # No parameters
    print("This")
    print("is")
    print("A")
    print("function")
# Function ended

# Calling the function in the program multiple times
my_print_function()
my_print_function()
```

Function in Python

Function Parameters

Parameters are a crucial part of the function structure.

They are the means of passing data to the function. This data can be used by the function to perform a meaningful task.

When creating a function, we must define the number of parameters and their names. These names are only relevant to the function and won't affect variable names elsewhere in the code. Parameters are enclosed in parentheses and separated by commas.

The actual values/variables passed into the parameters are known as arguments.

```
def minimum(first, second):  
    if (first < second):  
        print(first)  
    else:  
        print(second)
```

Function in Python

```
num1 = 10
```

```
num2 = 20
```

```
minimum(num1, num2)
```

Here, we are passing num1 and num2 to the function. The positions of the parameters are important. In the case above, the value of num1 will be assigned to first as it was the first parameter. Similarly, the value of num2 assigned to second.

If we call a function with lesser or more arguments than originally required, Python will throw an error.

A parameter can be any sort of data object; from a simple integer to a huge list

Function in Python

The return Statement

To return something from a function, we must use the return keyword. Keep in mind that once the return statement is executed, the compiler ends the function. Any remaining lines of code after the return statement will not be executed.

Let's refactor the minimum() method to return the smaller value instead of printing it. Now, it'll work just like the built-in min() function with two parameters:

```
def minimum(first, second):  
    if (first < second):  
        return first  
    return second
```

```
num1 = 10  
num2 = 20  
result = minimum(num1, num2) # Storing the value returned by the function  
print(result)
```

Function in Python

Scope Of Variables

All variables in a program may not be accessible at all locations in that program.

Part(s) of the program within which the variable name is legal and accessible, is called the scope of the variable. A variable will only be visible to and accessible by the code blocks in its scope.

There are broadly two kinds of scopes in Python –

- Global scope -In Python, a variable declared outside a function is known as a global variable. This means that a global variable can be accessed from inside or outside of the function.
- Local scope - Variables that are defined inside a function body have a local scope. This means that local variables can be accessed only inside the function in which they are declared.

Function in Python

The Lifetime of a Variable

The lifetime of a variable is the time for which the variable exists in the memory.

- The lifetime of a Global variable is the entire program run (i.e. they live in the memory as long as the program is being executed).
- The lifetime of a Local variable is their function's run (i.e. as long as their function is being executed).

Function in Python

Creating a Global Variable

The lifetime of a variable is the time for which the variable exists in the memory.

Consider the given code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

Here, we created a global variable `x = "Global Variable"`. Then, we created a function `foo` to print the value of the global variable from inside the function. We get the output as:

Global Variable

Thus we can conclude that we can access a global variable from inside any function.

Function in Python

What if you want to change the value of a Global Variable from inside a function?

Consider the code snippet:

```
x = 5
```

```
def foo():
```

```
    x = x - 1
```

```
    print(x)
```

```
foo()
```

In this code block, we tried to update the value of the global variable x. We get an output as:

UnboundLocalError: local variable 'x' referenced before assignment

Function in Python

In previous code block, we tried to update the value of the global variable x. We get an output as:

UnboundLocalError: local variable 'x' referenced before assignment

This happens because, when the command `x=x-1`, is interpreted, Python treats this x as a local variable and we have not defined any local variable x inside the function `foo()`. Is there anyway to fix this- say, we want to modify the variable inside function?

Function in Python

Use **global** keyword

What if you want to change the value of a Global Variable from inside a function?

Consider the code snippet:

```
c = 0 # global variable
```

```
def add():
```

```
    global c
```

```
    c = c + 2 # increment by 2
```

```
    print("Inside add():", c)
```

```
add()
```

```
print("In main:", c)
```

Function in Python

Creating a Local Variable

We declare a local variable inside a function. Consider the given function definition:

```
def foo():
```

```
    y = "Local Variable"
```

```
    print(y)
```

```
foo()
```

We get the output as:

Local Variable

Function in Python

Accessing A Local Variable Outside The Scope

```
def foo():  
    y = "local"  
    foo()  
print(y)
```

In the above code, we declared a local variable `y` inside the function `foo()`, and

then we tried to access it from outside the function. We get the output as:

NameError: name 'y' is not defined

We get an error because the lifetime of a local variable is the function it is defined in. Outside the function, the variable does not exist and cannot be accessed. In other words, a variable cannot be accessed outside its scope.

Function in Python

Global Variable And Local Variable With The Same Name

Consider the code given:

```
x = 5
def foo():
    x = 10
    print("Local:", x)
foo()
print("Global:", x)
```

In this, we have declared a global variable `x = 5` outside the function `foo()`. Now, inside the function `foo()`, we re-declared a local variable with the same name, `x`. Now, we try to print the values of `x`, inside, and outside the function. We observe the following output:

Local: 10

Global: 5

Function in Python

In the previous code snippet, we used the same name `x` for both global and local variables.

We get a different result when we print the value of `x` because the variables have been declared in different scopes, i.e. the local scope inside `foo()` and global scope outside `foo()`.

When we print the value of the variable inside `foo()` it outputs Local: 10. This is called the local scope of the variable. In the local scope, it prints the value that it has been assigned inside the function.

Similarly, when we print the variable outside `foo()`, it outputs global Global: 5.

This is called the global scope of the variable and the value of the global variable `x` is printed

Default arguments in Python

- Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value.
- Default Arguments:
- Python has a different way of representing syntax and default values for function arguments.
- Default values indicate that the function argument will take that value if no argument value is passed during the function call.
- The default value is assigned by using the assignment(=) operator of the form `keywordname=value`.

Default arguments in Python – Calling without keyword argument

```
def student(firstname, lastname='Mark', standard='Fifth'):
    print(firstname, lastname, 'studies in', standard, 'Standard')
```

1 positional argument

```
student('John')
```

3 positional arguments

```
student('John', 'Gates', 'Seventh')
```

2 positional arguments

```
student('John', 'Gates')
```

```
student('John', 'Seventh')
```

Default arguments in Python – Calling with keyword argument

```
def student(firstname, lastname='Mark', standard='Fifth'):
    print(firstname, lastname, 'studies in', standard, 'Standard')
```

1 keyword argument

```
student(firstname='John')
```

2 keyword arguments

```
student(firstname='John', standard='Seventh')
```

2 keyword arguments

```
student(lastname='Gates', firstname='John')
```

Variable number of arguments in Python function

Say, you want to multiply integer numbers and return result from a Python function.

```
def multiplythreenumbers(num1,num2,num3):  
    return num1*num2*num3  
multiplythreenumbers(2,3,4)
```

What if I want to multiply 4 numbers and not three as shown above?

Another function multiplyfournumbers?

How about five arguments? multiplyfivenumbers? No, this is not an elegant solution.

Best way to solve this is to use variable number of arguments

Variable number of arguments in Python function

Using a function that accepts a variable number of arguments can be very useful: this provides a lot of flexibility and reduces the clutter in the function signature. Besides, it does not make any assumptions about the number of needed arguments, which can be appropriate in multiple scenarios.

Using ***args** in a function is Python's way to tell that this one will:

- Accept an arbitrary number of arguments
- Pack the received arguments in a tuple named args. Note that args is just a name and you can use anything you want instead.

Variable number of arguments in Python function

```
def multiply_numbers(*numbers):  
    product = 1  
    for number in numbers:  
        product *= number  
    return product
```

This function can now receive an arbitrary number of arguments and even if you have a list of numbers, you can still use it

Function Assignment

1.You must implement the rep_characters function. You are given two integers, x and y, as arguments. You must convert them into strings. The string value of x must be repeated 10 times and the string value of y must be repeated 5 times.

```
def rep_cat(x, y):  
    return str(x) * 10 + str(y) * 5
```

Lambdas – Anonymous functions in Python

We've always given names to our functions using the `def` keyword. However, there is a special class of functions for which we do not need to specify function names.

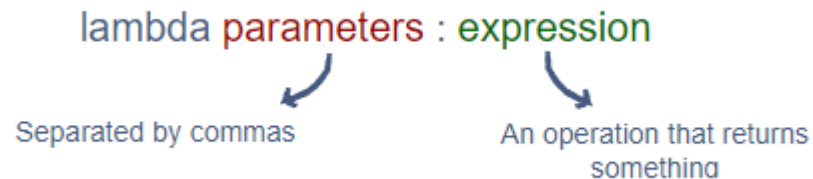
A **lambda** is an **anonymous** function that returns some form of data.

Lambdas are defined using the `lambda` keyword. Since they return data, it is a good practice to assign them to a variable.

The following syntax is used for creating lambdas:

In the structure below, the parameters are optional.

`lambda parameters : expression`



The diagram illustrates the syntax of a lambda function: `lambda parameters : expression`. A blue arrow points from the text "Separated by commas" to the `parameters` part of the syntax. Another blue arrow points from the text "An operation that returns something" to the `expression` part of the syntax.

Separated by commas

An operation that returns something

Lambdas – Anonymous functions in Python

```
triple = lambda num : num * 3 # Assigning the lambda to a variable
```

```
print(triple(10)) # Calling the lambda and giving it a parameter
```

Here's a simple lambda that concatenates the first characters of three strings together:

```
concat_strings = lambda a, b, c: a[0] + b[0] + c[0]  
print(concat_strings("World", "Wide", "Web"))
```

Lambdas – Anonymous functions in Python

As we can see, lambdas are simpler and more readable than normal functions. But this simplicity comes with a limitation.

Python lambdas are only a shorthand notation if you're too lazy to define a function

A lambda cannot have a multi-line expression. This means that our expression needs to be something that can be written in a single line.

Hence, lambdas are perfect for short, single-line functions.

We can also use conditional statements within lambdas:

When using conditional statements in lambdas, the if-else pair is necessary. Both cases need to be covered, otherwise, the lambda will throw an error:

```
my_func = lambda num: "High" if num > 50 else 'low'
```

Lambdas – Anonymous functions in Python

traditional function with def

```
def sum(a, b):  
    return a + b  
print(sum(1, 2))
```

lambda function with name

```
sum_lambda = lambda x, y: x + y  
print(sum_lambda(1, 2))
```

3

Let's check the type of two functions-

```
print(type(sum))  
print(type(sum_lambda))
```

Functions as Arguments

In Python, one function can become an argument for another function. This is useful in many cases. In other words, higher order functions can take functions as input params or return function as output or both.

Functions as Arguments

Use Lambda function with Python higher-order functions

Lambda functions are normally combined with Python higher-order function.

A higher-order function is a function that does at least one of the following:
a) takes one or more functions as arguments. b) returns a function as its result.

In Python, there are a couple of built-in higher-order functions like map, filter, reduce, sorted, sum, any, all. Among them, map, filter, and reduce are being used most often with lambda functions.

Use Lambda function with Python higher-order functions

Map

map() function takes 2 arguments: an input mapping function and an iterable. The mapping function will be applied to each element in the iterable. map() function returns an iterator containing the mapped elements.

The mapping function can be represented using a clean and short lambda function.

```
def mapping(x):  
    return x**2
```

```
mapped_def = list(map(mapping, [1,2,3]))  
print(mapped_def)  
# [1,4,9]  
mapped_lambda = list(map(lambda x:x**2, [1,2,3]))  
print(mapped_lambda)  
# [1,4,9]
```

Use Lambda function with Python higher-order functions

Filter

`filter()` function takes the same arguments as `map()`: an input filtering function and an iterable. The filtering function will be applied to each element and `filter()` function returns an iterator containing the filtered elements.

Here is an example to filter even numbers using `def` function and `lambda` function.

```
def filter_func(x):  
    return x % 2 == 0
```

```
filter_def = list(filter(filter_func, [1, 2, 3]))  
print(filter_def)  
filter_lambda = list(filter(lambda x: x % 2 == 0, [1, 2, 3]))  
print(filter_lambda)
```

Use Lambda function with Python higher-order functions

Reduce

`reduce()` function is from Python built-in module `functools`. What it essentially does is to cumulatively apply a function to all the elements in an iterable, and generate a single value. `reduce()` function takes 3 arguments: an input function, an iterable and an optional initializer.

```
def sum(x, y):  
    return x + y
```

```
print(reduce(sum, [1, 2, 3]))
```

```
print(reduce(lambda x, y: x + y, [1, 2, 3]))
```

Lambda function assignment

1. Write a lambda function to double the elements of a given list. Use `map()` function.
2. Write a lambda to filter elements greater than 10 from a given list. Use `filter()` function.
3. Write Fibonacci program using Lambda function

Solution

```
1. num_list = [0, 1, 2, 3, 4, 5]
```

```
double_list = map(lambda n: n * 2, num_list)
```

```
print(list(double_list))
```

```
2. numList = [30, 2, -15, 17, 9, 100]
```

```
greater_than_10 = list(filter(lambda n: n > 10, numList))
```

```
print(greater_than_10)
```

```
3. fib = lambda x: x if x <= 1 else fib(x-1) + fib(x-2)
```

Type hinting and annotations

- Python is a dynamically typed language, meaning that there is no static type checking involved unless the code is executed. This can lead to runtime bugs that are difficult to debug and fix. Hence, type hints, also called type annotations, were introduced in Python 3.5 via the typing module to support the external static type checkers and linters to correctly recognize errors.
- It is relatively new feature in Python.
- The major benefit of having type hints in your codebase is about future maintenance of the codebase.
- When a new developer will try to contribute to your project, having type hints will save a lot of time for that new person.
- It can also help to detect some of the runtime issues we see due to passing of wrong variable types in different function calls.

Type hint for built-in types

Built type	Type hint
<code>int</code>	<code>int_var: int = 3</code>
<code>float</code>	<code>float_var: float = 3.4</code>
<code>boolean</code>	<code>bool_var: bool = True</code>
<code>string</code>	<code>str_var: str = "edpresso"</code>

Type hinting for collection types

The type hints for collections in python are to be imported from the typing module.

from typing import List, Set, Dict, Tuple

Collection	Type hint
list	list_int: List[int] = [3]
list	list_string: List[str] = ["edcoffee"]
set	set_int: Set[int] = {9,1,3}
tuple (finite number of elements)	tuple_float: Tuple[float, str, int] = (3.5, "educative", 9)
tuple (infinite number of elements)	tuple_float: Tuple[float, ...] = (3.5, 2.3, 45.6)
dict	dict_kv: Dict[str, int] = {"key": 4}

Type hinting and function annotations

If we use Type hinting while writing function definition(argument and return type both have type hint)–

```
def add(a: int, b: int) -> int:  
    return a + b
```

```
print(add(5,8))
```

```
print(add("Futureense","Technologies"))
```

```
⚠ Expected type 'int', got 'str' instead :4
```

```
⚠ Expected type 'int', got 'str' instead :4
```

Note:You can see that the return type of the function call is defined after ->

Function in Python

Best Practices

- It is a good practice to define all our functions first and then begin the main code. Defining them first ensures that they can be used anywhere in the program safely.
- Write meaningful for the custom function created
- Return proper value from the function (if required)
- Provide proper comments while writing function – number of parameters ,return type etc
- Try to make the code as modular as possible by writing function. Don't ,however, over do it.

Python docstrings

Python docstrings are strings used right after the definition of a function, method, class, or module (like in Example 1). They are used to document our code.

We can access these docstrings using the `__doc__` attribute.

Whenever string literals are present just after the definition of a function, module, class or method, they are associated with the object as their `__doc__` attribute. We can later use this attribute to retrieve this docstring.

Python docstrings

```
def square(n):  
    """Takes in a number n, returns the square of n"  
    return n**2
```

```
print(square.__doc__)
```

Takes in a number n, returns the square of n

Python docstrings

Assignment

1. docstrings for the built-in function `print()`:
2. docstrings for the built-in function `min()`:
3. docstrings for the built-in function `len()`: