Python Tutorial

Welcome to your new knowledge base space!

- We've added some suggestions and placeholders. Everything is customizable.
- Get started with page templates:
  - ℹ️ Template - How-to guide
  - 🧐 Template - Troubleshooting article

Welcome! How can we help you?

Search our help articles

About

Add a little introduction about what you or your team does, what information this space provides, and any additional context.

Get in contact

- Team@email.com
- #slackchannel
- Raise a ticket at <link>

Top topics

🍓 Topic 1
- Link to page
- Link to page
- Link to page

🥝 Topic 2
- Link to page
- Link to page
- Link to page

🍍 Topic 3
- Link to page
- Link to page
- Link to page

🍐 Topic 4
- Link to page
- Link to page
- Link to page

🍋 Topic 5
- Link to page
- Link to page
- Link to page

🍒 Topic 6
- Link to page
- Link to page
- Link to page

Team name

Get Support
- Link to page
- Link to page
- Link to page

Related Links
- Link to page
- Link to page
- Link to page

Template - How-to guide

Instructions

1.
2.
3.

> Highlight important information in a panel like this one. To edit this panel's color or style, select one of the options in the menu.

Related articles

🧐 Template - Troubleshooting article

🤔 Problem

🌱 Solution

1.
2.
3.

> Highlight important information in a panel like this one. To edit this panel's color or style, select one of the options in the menu.

📎 Related articles

Chapter 1: Introduction to Python

Introduction to

Python is a high level, portable and multi-paradigm programming language, developed by Guido van Rossum and released in 1991. Python comes with powerful and efficient inbuilt data structures along with object oriented programming support.

Why should we learn Python

Python has gained its popularity very quickly and has high demand in cutting edge technologies. One can easily start his or her career with this language. Some of the key features of Python are listed below:

- **Open Source:** Python is open source; hence it is free to use and distribute.
- **Multi-Platform Support:** Python is supported by Windows, Mac, Linux, Raspberry Pi, etc. that makes it easier to use on different devices.
- **Easy to Learn:** Python has English like syntax and it is super simple to learn. Even a beginner in programming can learn python very easily.
- **Multiparadigm:** Python supports Object Oriented, functional, procedural and imperative programming paradigms.
- **Scientific Calculations:** Python supports very large calculations and it can be used in scientific calculations and data visualization.
- **Rich Libraries:** Python has a large library support. Almost everything can be done using python due to a vast library support.

Uses of Python:

Python was developed as a scripting language, but its easiness has extended it to many other tasks. Python is currently being used for:

- Web development
- Data Science and Machine Learning
- Artificial Intelligence
- Automation and Scripting
- Desktop Application
- Data Analysis and Visualization

Apart from these, Python is able to do general programming as well as it constantly being improved to support mobile devices and desktop application development.

Comparison with Other Languages:

- Python has powerful data structures that are available without importing them. Python supports list, set and dictionaries that are often not available in other languages implicitly.

- Python relies on indentation to create blocks. This makes it easy to read and makes the code beautiful. With other languages, the code becomes messy after some revisions but that is not the case in Python.
- Python can calculate large numbers which is an issue in other languages. Most of the languages support up to 64 or 128 bit numbers.
- Semicolons are optional in Python. Python depends on new line characters to distinguish between statements. It makes it easy to type and programmers do not need to worry about semicolons.
- Python usually provides similar functionality in very few lines of code. This makes it pretty useful in rapid prototyping and automation.

Conclusion:

Python is simple yet powerful. Python has been one of the most popular languages in past years. Hence it is a must have language for your resume.

Chapter 2: Python Installation & Getting Started

Python Installation

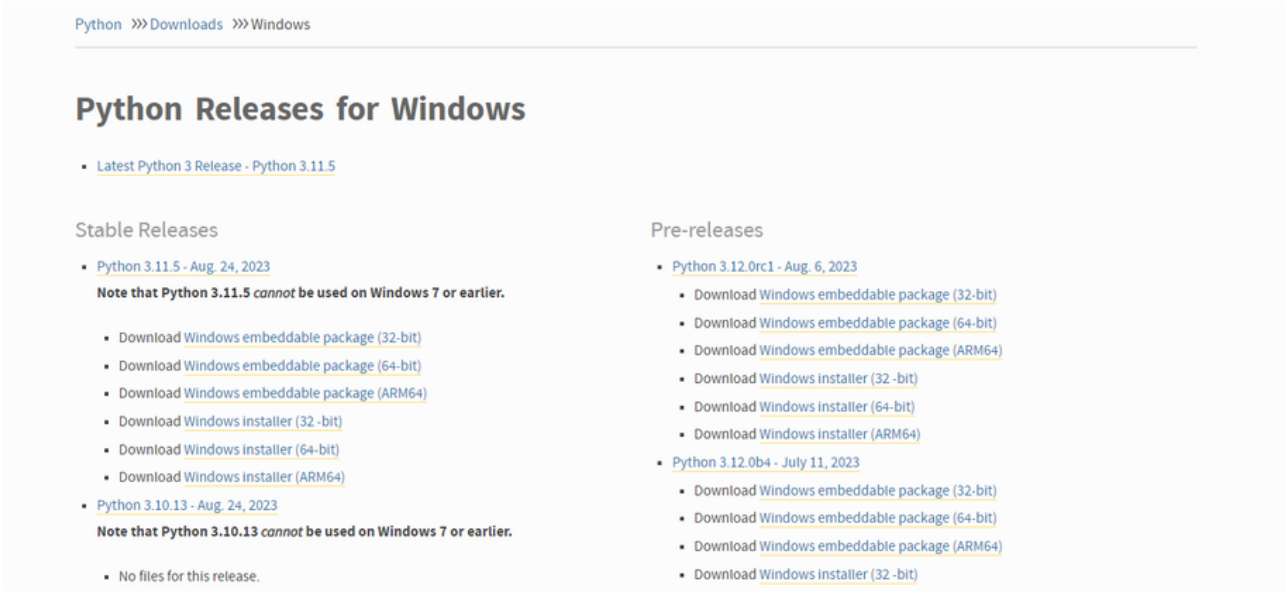
Installing python on your machine is a piece of cake. Just make sure that you are connected to internet and we need to follow some simple steps.

Installation on Windows:

Windows does not come with python preinstalled. So we need to manually install it. Follow these steps to install python on your windows machine:

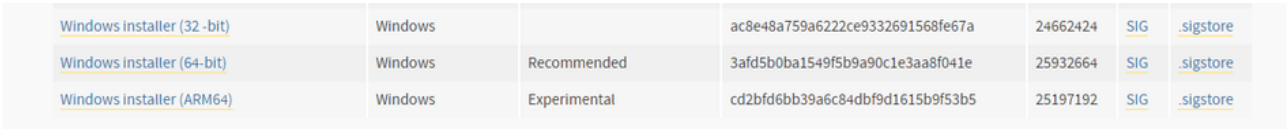
- Open your browser and go to https://python.org and hover on the downloads option as highlighted in this snapshot.

- Choose windows and click on the latest python version. This will open a new page where all the recent python versions are listed.

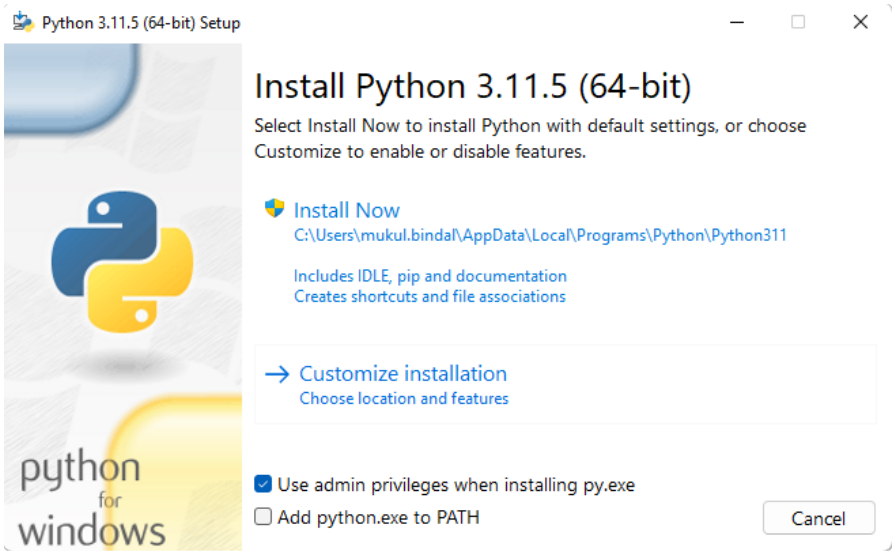


- Choose the latest version available. In this snapshot, Python 3.11.5 is the latest version available but it may change in future.

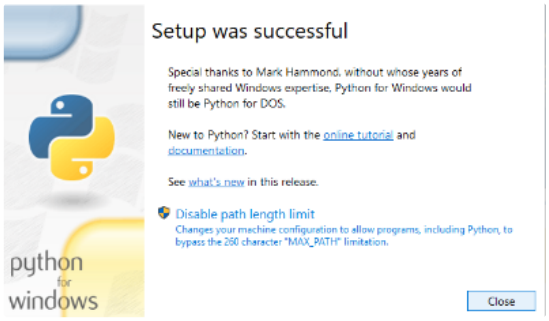
- Now click on the correct installer as per your device configuration. This will download an executable file that you can install on your machine.



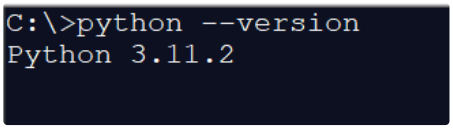
- Open the installer by double clicking on it. Make sure that you check the Add Python to Path option. This option will automatically set the environment variables for you.

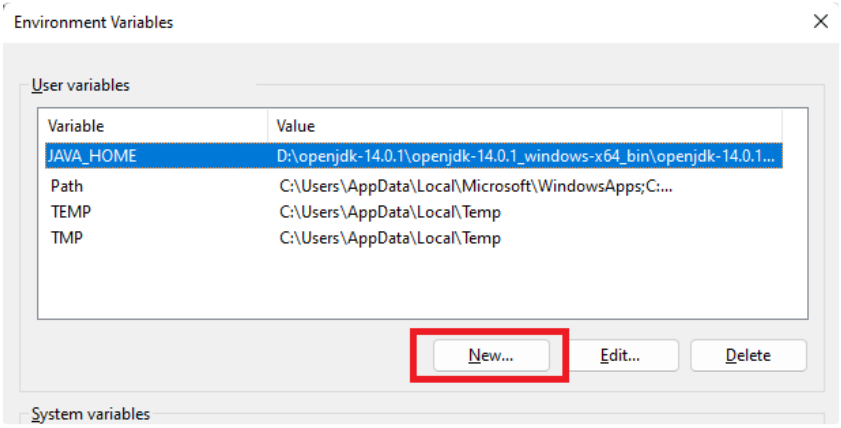


- Now click on install now option and wait for 1-2 minute while it is installing.
- You will see the following screen once setup is successful. If you don't, try repeating the steps again.

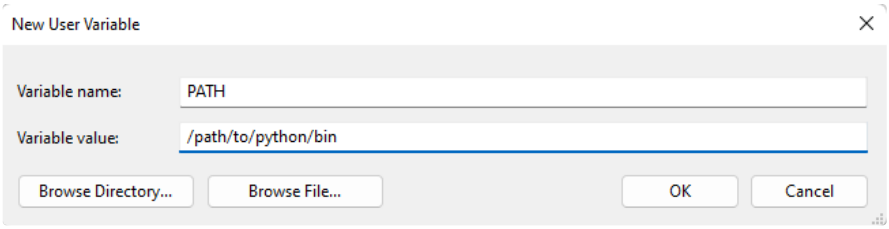


To cross check whether it is installed or not, open command prompt, type `python --version` and hit enter. This should display the current python version without any errors.

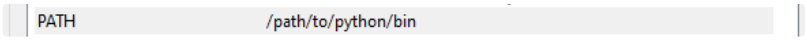


- If you are using an older version of Python, the environment variables may not set automatically.
- Hence we need to set them manually:
  - Type "Environment Variables" in search bar of windows. Open "Edit the system environment variables".
  - 

- Click on “new..”.



- Type the Variable Name and Variable Path and click on “OK“ and path is set successfully.



Installation on Linux:

Most of the Linux distributions come with Python preinstalled. Open the terminal using `Ctrl+Alt+T` and type `python3 --version` and it should show the version details. If python is not installed, run the following commands to install python in your machine:

- Install the supporting software as it might be missing from your machine.

```
sudo apt update
sudo apt install software-properties-common
```

- Add the `deadsnake PPA` if required, as it contains more updated and latest binaries.

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
```

- Finally install Python using below command.

```
sudo apt install python3.9
```

Our First Script in Python:

Now we have python installed in our system, so we can write our first script. We can write our script in any text editor like Notepad in Windows or Leafpad in Ubuntu. But for a large scale project, we should use a multifeatured IDE for easy development. Here are some recommended IDEs for python that are popular these days:

1. Sublime Text 3 : This is an all in one IDE that is super light and has rich collection of plugins.
2. VS Code : VS code is a Microsoft product and it has a free version available. It supports many environments with its large set of extensions.
3. PyCharm : PyCharm is an integrated development environment designed specifically for the Python language.
4. Atom : Atom is free and open source IDE with plugins written in JavaScript. It is the most loved IDE for MacOS and Linux.
5. Spyder and Jupyter : For data science, machine learning and Data Visualization, these IDEs are the best ones. They also have their free and paid versions available.

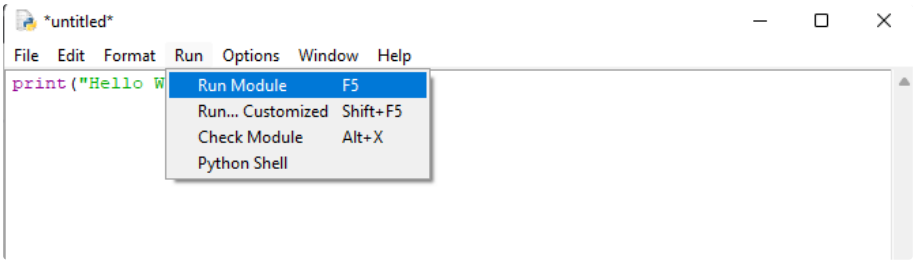
For now we can either grab one of these or we can simply use python’s inbuilt ide called IDLE. It is installed by default in Windows at the time of python installation.

Create a new file named `hello.py` and make sure that file name ends with `.py` only. `.py` is the extension for python scripts. Add the following line inside the file using any of the text editor or IDE.

```
print("This is my first python program")
```

Now save the file and run the script. We can run the script in two ways:

1. **Use the IDE:** To run the program, all IDEs provide some options in GUI. We can press `F5` in IDLE or click on green play button on VS Code. You should check your IDE documentation on how to run a program.



2. **Using command line(Recommended):** Open command prompt or terminal in the directory where your hello.py file resides. Type `python hello.py` and hit enter. You should see the following output in the console:

```
1 This is my first python program
```

The above program uses a inbuilt function `print()`, that takes the given string as parameter and logs it on console. We will see the details about this function in later chapters.

Interactive Shell of Python:

Python is an interpreted language. It means that the script is not complied beforehand but it is executed line by line. Python provides us with an Interactive shell where users can type the commands one by one and get the live response. This feature allows us to test certain features and to make small prototypes before writing the main program. Follow the steps below

Open the “command prompt”. Type `python` and hit `Enter`.

```
C:\>python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You will see a new environment with `>>>` in each line. Here users can write the commands. Type any arithmetic expression and hit `enter.`

```
C:\>python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 20 + 30
50
>>>
```

You can see that we get the immediate response. Now type `print(“Hello World“)`. You will see the expected output.

```
C:\>python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 20 + 30
50
>>> print("Hello World")
Hello World
>>>
```

Congratulations, you have just stepped into the world of python by running your very first script.

In this tutorial we have learned how to install python on your system as well as we ran our first python script using IDE, command line and Interactive shell. There are many more exciting things coming in next chapters.

Chapter 3: Variables and Objects

When we write a program, we basically perform a series of operations on some data. Our program contains the data and we process it to get some useful information. Here, the data is provided to program in form of **input**. It is stored inside a **variable or object**. The process is nothing but applying operations among the variables using **operator**. And finally the useful information we get is our **output**.

In this chapter, our goal is to understand what are variables and objects, how they work and how we use them in python.

Variables:

When the data is loaded into primary memory, we need to use a name or symbol in order to access or modify it. Variables are the virtual containers that hold the data and provide us a medium to manipulate it. A variable can hold the address of the data stored in memory.

Objects:

Objects are real life entities that have a particular state, an identity and behavior. In python, almost everything is an object. Hence the data resides in the object but it is symbolically represented through a variable.

> *We will learn about objects and classes in detail in upcoming chapters.*

Variable Declaration:

In python, we don't need to declare variables with their datatypes. Python automatically detects the datatype or class using the initial value. We use the assignment operator "=" to assign the initial value to a variable.

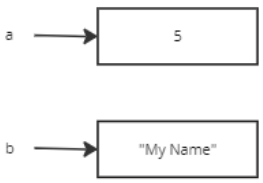
Try the following code:

```
1 a = 5
2 b = "My Name"
3 print(a)
4 print(b)
```

Output:

```
1 5
2 My Name
```

- In line 1, we have assigned the value 5 to variable `a`. Python automatically detects that it is an integer and creates an integer type object with value 5.
- In line 2, we have assigned the string to variable `b`. Python creates a new string object and assigns its address to variable `b`.



- In line 3 & 4 we have printed the values of the variables on the console using `print()` function that we used previously too.

Variable Naming conventions in Python:

There are certain rules that we must follow when we choose a variable/object name. These rules are:

- A variable name can start with either a letter (A-Z or a-z) or an underscore (_).
- A variable name can only contain alphanumeric characters (A-Z, a-z and 0-9) or the underscore character only.
- A variable name should not be a keyword. (See the table below)
- Variable names are case sensitive. Hence the variables `animals`, `Animals` and `ANIMALS` all are different.
- With above points, `age`, `Python`, `_hello`, `Game007` all are valid variable names while `3name`, `hello@world`, `123xyz`, `$var` all are invalid variables.

Below are the reserved keywords that we should not use as variable names in python:

Keywords in Python		
and	as	assert
async	await	break
class	continue	def
del	elif	else
except	finally	for
from	global	if
import	in	is
lambda	nonlocal	not
or	pass	raise
return	try	while
with	yield	True
False	None	

> Can you identify the invalid variable names from the following?
>
> `_hello`, `404error`, `maxLength`, `raise`, `stack_size`, `highest value`, `amount_in_$`, `in`

type() and id() functions:

- **type()** is an build function just like `print()`. `type()` returns a value that is the class name of the object or variable that we pass in it.

```
1 a = 5
2 name = "Python"
3 print( type(a) )
4 print( type(name) )
```

Output:

```
<class 'int'>
<class 'str'>
```

From the above output, we can see that variable `a` is `int` variable that is the class of Integers. And `name` is `str` variable that is the implicit String class.

- **id()** is also an inbuilt function that returns the physical memory address of the variable and objects that we pass in it. We have discussed before that each value resides in main memory at a particular memory address. `id()` function returns us the address.

```
a = 10
print( id(10) )
```

Output:

```
1965805824432
```

This big number is the address of the value in the memory.

We will see more inbuilt functions where ever they are required.

Multiple variable assignment:

Python supports inline multiple variable assignment. This is one of the unique features of python. Observe the below syntax:

Syntax:

> variable1, variable2, … ,variableN = value1, value2, …, valueN

See the following code for better understanding.

```
a, b = 1, 2
x, y, z = 3, 4, 5
print(a)
print(b)
print(c)
print(x)
print(y)
print(z)
```

Output:

```
1
2
3
4
5
```

We can see that we can declare multiple variables in a single line instead of declaring each of them separately.

Comments in Python:

Comments are the lines that are skipped while execution. We use comments to disable some part of our code as well as to write description about what our code is about and how is it working. In python, we can use `#` symbol before any statement to make it a comment.

```
# Demonstration of comments
print(10)
```

```
# print(15)
```

Output:

```
10
```

We can observe that line 1 has no effect on the code even if it is just an English statement. Line 2 is executed and the result is printed while line 3 is not executed because it is treated as a comment due to `#` symbol in the start.

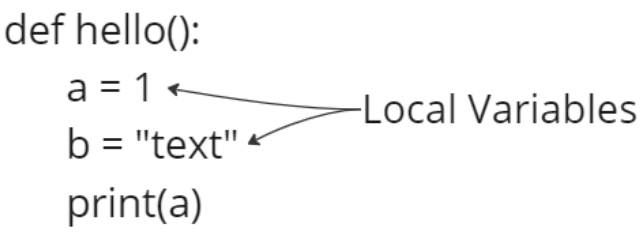
> As a good programming practice, we should always use comments in our code. Comments are not necessary in each line, but they should be used in complex parts and should be brief.

Local and Global variables:

Python uses 1 tab indentation to create a block of code. See the code below:

Code Block

```
def hello():
  a = 1
  b = "text"
  print(a)

```



> In the above code snippet, line 1 is the function definition and `def` is the keyword to create a user defined function.
>
> We will see details about user defined function in later chapters. For now, we use `def function_name()` to create a function and `function_name()` to call it just like we were calling `print()`.

> We can see that there is no `{}` to create the block of this function. Python uses `:` to indicate that a block of code has to start in the next line. After that we write the code with 1 tab indentation. Here, line 2, 3 and 4 are part of this block.

The variables declared inside a block are called **local variables.** Here `a` and `b` are local variables.

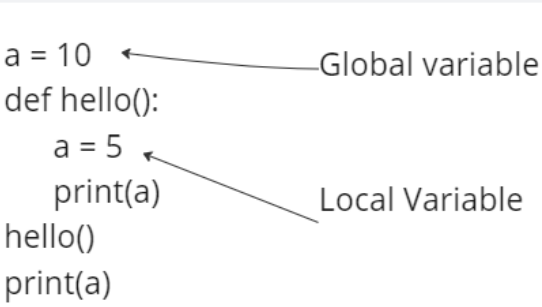
The variables that are not in blocks, are called **global variables** and they can be accessed from anywhere.

Now, see the code below:

Code Block

```
a = 10
def hello():
    a = 5
    print(a)
hello()

```

```
print(a)

```



Output:

Code Block

```
5
10

```

Here, variable `a` in line 1 is global variable as it is not in a block. But variable `a` in line 3 is local variable. When we try to access `a` in line 5, it will access the local `a` with value `5` but at line 6, we can not access the local variable `a`. Hence it will print value of global variable with value `10`.

> We will learn more about global variables and their application in scopes chapter.

Object Reference:

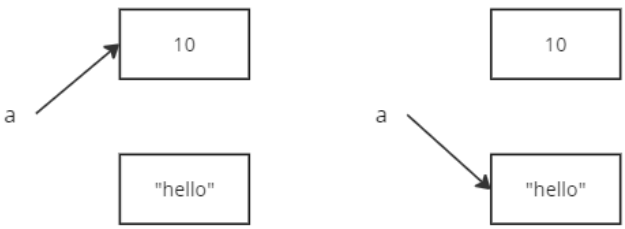
Whenever we declare a variable, say `a = 100`, the following events occur internally:

1. Python creates an object of class `int` with value 100.
2. A variable with name `a` is created.
3. The object is mapped with symbol `a` so that we can access or modify it.

This internal processing is called object referencing and because of this, python does not bind a variable with the data type. If we create an `int` variable and later assign it a `str` value, this will not create any error. Hence the below code will run perfectly:

Code Block

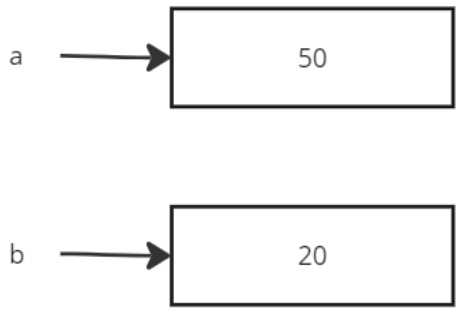
```
a = 10
a = "hello"

```



Lets do a little exercise to understand better. Type the following code in a file:

Code Block

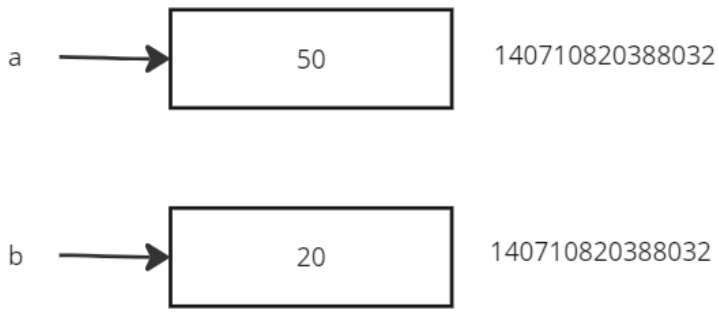
```
a = 50
b = 20
print(id(a))
print(id(b))

```



Output:

Code Block

```
140710820388032
140710820387072

```

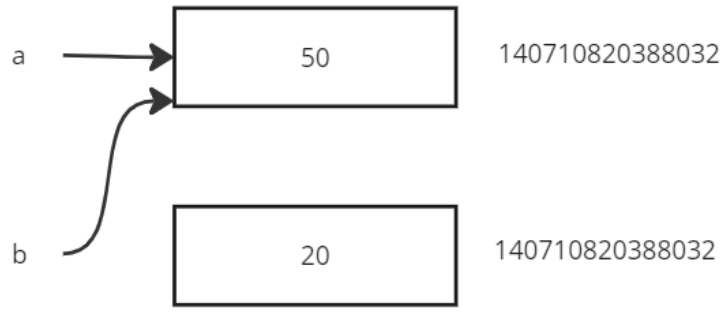


Here two different objects are created in the memory and hence they have different ids. Note that the ids can be different on your machine as they are physical addresses.

Now if we assign `a` to `b`, the variable `b` will simply start pointing to the object of `a`.

Code Block

```
b = a
print(id(a))
print(id(b))

```



Output:

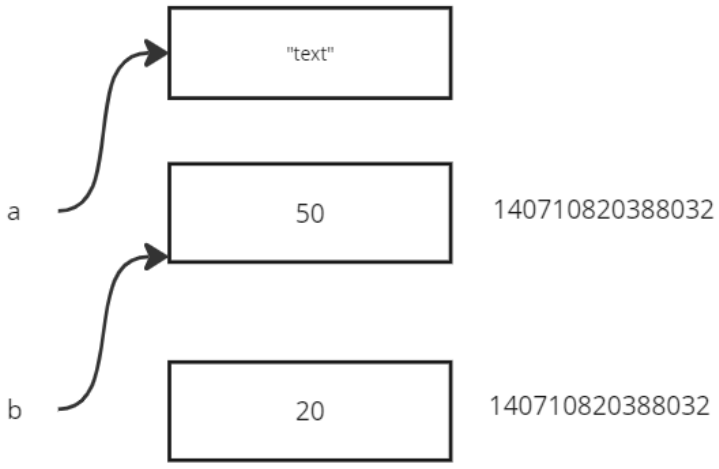
Code Block

```
140710820388032
140710820388032

```

You can see that both the variables are now pointing to the same memory location. If we assign a new value to `a`, it will start referencing the new object while `b` will still point to the same object.

Code Block

```
a = "text"
print(id(a))
print(id(b))

```



Output:

Code Block

```
1825716070448
140710820388032
```

In this tutorial, we have learned how to declare a variable, how to choose a proper name for it and what are the reserved keywords in python. We also learned a unique feature of multiple variable assignment as well as comments , global variables and object reference in python. In the next chapter, we will learn about the data types available in python and how we can use them in our programs.

Chapter 4: Data Types

Our program deals with different types of data like text, numbers, sequences and many other types. Python has built in data types along with some powerful data structures. In this chapter, we will try to understand the basic data types of python in details.

Types of data:

As we have mentioned before, in python, almost everything is an object. It applies for data types also. The data types are also classes. A class is simply a blueprint or structure of how a real life entity behaves.

Below are the data types available in Python:

Type of Data	Available data types / classes	Examples
Numbers	int, float, complex	10, 3.1415, 32000, 7+8j
Text	str	“This is some text“, 'hello there'
Boolean	bool	True, False
Sequences	list, tuple	[1,2,4], (3,3,6)
Set and Maps	set, dict	{1,2,4} {'a':1, 'b':9}
Binary Data	bytes, bytearray, memoryview	b”some data here”, b'\x01\x02\x03'
Null data	NoneType	None

Number Data Type:

1. Integers:

Integers are the whole numbers which can be positive, negative or 0 without any fractional part. For example, 5, 10, 45 are integers while 3.6, 4.0 are not integers. In python, `int` class is used to represent integers. There is no limit on how big the integer can be.

```
a = 100
print(type(a))
```

Output:

```
<class 'int'>
```

2. Floating point numbers:

Floating point numbers can contain the fractional part too. Python recognizes the numbers with fraction as float values. Python has `float` class to represent the floating point numbers. We can use `float` constructor to convert any valid value to float.

```
a = 100.0
b = float(100)
print(type(a))
print(type(b))
```

Output:

```
<class 'float'>
```

```
<class 'float'>
```

3. Complex Numbers:

Python also support an additional data type for complex numbers. We can write the imaginary part with a following `j` character. Python has inbuilt `complex` class to represent complex values. We can pass the real and imaginary part in `complex` constructor to create a complex number.

```
a = complex(1,2)    # First way
b = 4+7j            # Second way
print(a)
print(b)
print(type(a))
```

Output:

```
(1+2j)
(4+7j)
<class 'complex'>
```

Text Data Type:

In programming, text data type is known as `string`. Python has inbuilt `str` class to support strings. We can enclose any text within single quotes `' '` , double quotes `" "` as well as triple quotes `''' … '''` and python will detect it as an instance of string class. Python doesn't have `char` datatype, hence for representing single characters, we use `str` class only.

```
s = "this is some text"
r = 'this is another text'
print(type(s))
print(type(r))
```

Output:

```
<class 'str'>
<class 'str'>
```

Boolean Data Type:

A boolean data type has only 2 values, either `true` or `false`. Python has inbuilt `bool` class to represent boolean data. The available values are `True` and `False`.

```
t = True
f = False
print(type(t))
print(type(f))
```

Output:

```
<class 'bool'>
<class 'bool'>
```

Sequence Data Types:

Python supports some data structures to hold a collection of data. A sequence is the ordered collection of data. Python has inbuilt `list` and `tuple` classes to store sequential data. Both of the classes support heterogeneous data.

- A `list` can be declared using comma separated values enclosed in square brackets `[ ]`. We can also pass an iterable to get the corresponding list. See the example below:

```
l = [1,2,'three', 4.0]
print(type(l))
```

Output:

```
<class 'list'>
```

- A `tuple` is also a sequence. Just like list, we enclose the values in round brackets `( )`.

```
t = (1,2,'three', 4.0)
print(type(t))
```

Output:

```
<class 'tuple'>
```

> We will see the detailed data structures in their corresponding chapters.

Set Data Type:

Python supports set using `set` class. A set is unordered and it contains unique values only. We represent set as comma separated values enclosed in flower brackets `{ }` .

```
s = {1,3,5}
print(type(s))
```

Output:

```
<class 'set'>
```

Map Data Type:

Maps are known as ‘dictionary’ in Python. Dictionaries are available in the form of `dict` class. A `dict` object contains key value pairs where all the keys are unique. We can access the values using their keys.

> *Syntax:*
>
> *d = { key1 : value1, key2 : value2, … , keyN : valueN }*

See the example below:

```
d = {'a':1, 4:8, "animal":9}
print(type(d))
```

Output:

```
<class 'dict'>
```

> We will see all the methods of set and dictionary in the upcoming chapters.

Binary Data Types:

Python supports 3 types of Binary data types:

1. **bytes:** `bytes` objects are sequence of single bytes that are immutable. Immutable means they can be read but we can not modify them.
2. **bytearray:** `bytearray` objects are also single byte sequences but they are mutable.
3. **memoryview:** `memoryview` objects let us deal with the internal data or the buffer. We can programmatically access and modify the raw data using `memoryview` objects.

```
bytes_object = bytes(b'encoded data')
print(type(bytes_object))

bytearray_object = bytearray(b'encoded data')
print(type(bytearray_object))

memoryview_object = memoryview(b'encoded data')
print(type(memoryview_object))

```

Output:

```
<class 'bytes'>
<class 'bytearray'>
<class 'memoryview'>
```

Here the preceding `b` denotes the bytes literal class.

Null Data Type:

Python has a `NoneType` class to support Null data. The `None` keyword denotes the null data.

```
print(type(None))

```

Output:

```
<class 'NoneType'>
```

Chapter 5: Operators in Python

Operators are the basic operational units of every language. The raw input data is processed by applying a series of operations to convert it into useful information known as output. These operations are applied with the help of operators. In this chapter, we will learn more about the operators and their application in our programs.

Categories of Operators:

On the basis of their functionalities, operators are categorised as follows:

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Membership Operators
7. Identity Operators

We will discuss each category in details with some examples.

Arithmetic Operators:

These operators are used for basic calculations like addition, multiplication etc.

Operator Name	Symbol	Description	Example	Result
Addition / Concatenation	**a + b**	Adds a and b Concatenates two strings a and b	9+7 “hello“ + “world“	16 “helloworld“
Subtraction	**a - b**	Substracts b from a	20 - 6	14
Multiplication	**a*b**	Multiplies a and b Concatenates string a, b number of times if b is integer	7*8 “hello“*3	56 “hellohellohello“
Division	**a/b**	Performs float division a and b	20/8	2.5
Integer Division	**a//b**	Returns floor of a/b	30//7	4
Modulus	**a%b**	Returns remainder when a divided by b	20%6	2
Power	**a**b**	Return a raise to power b	4**2	16

Note : The `/` operator always return `float` result, no matter the division returned non fractional number or not. Hence we should use `//` whenever `int` result is required.

Tip: The `%` operator is used to check divisibility too. If `b` can divide `a`, `a%b` will result in 0.

Comparison Operators:

Comparison operators are used to compare two objects and they return either True or False.

Operator Name	Symbol	Description	Example	Result
Equals	**a == b**	Returns True if a and b are equal	8 == 8	True
Greater Than	**a > b**	Returns True if a is greater than b	6 > 10	False
Less Than	**a < b**	Returns True if a is lesser than b	6< 10	True
Greater than or equals to	**a >= b**	Returns True if a is either greater than or it is equal to b	7>=7	True
Less than or equals to	**a <= b**	Returns True if a is either lesser than or it is equal to b	8<=10	True
Not equal to	**a != b**	Returns True if a is not equal to b	8 != 9	True

---

Logical Operators:

These operators are used in conditional statements to combine boolean expressions.

Operator Name	Symbol	Description	Example	Result
Logical AND	**a and b**	Returns True if a is True and b is also True. Otherwise returns False	5>0 and 6>5	True
Logical OR	**a or b**	Returns False if a and b both are False. Otherwise returns True	7<0 or 6>0 7<0 and 8<5	True False
Logical NOT	**not a**	If a is False it returns True, otherwise it returns False	not 6>0	False

---

Bitwise Operators:

Bitwise operators are used to apply boolean operations on binary numbers at machine level.

Operator Name	Symbol	Description	Example	Result
Bitwise AND	**a & b**	Performs bitwise AND between a and b	6 & 5	4
Bitwise OR	**a | b**	Performs bitwise OR between a and b	4 | 3	7
Bitwise NOT	**~a**	Performs bitwise NOT on a	~5	-6
Bitwise XOR	**a ^ b**	Performs bitwise XOR between a and b	5 ^ 8	13
Left Shift	**a << b**	Left shifts bits of a by b number of places	5 << 2	20
Right Shift	**a >> b**	Right shifts bits of a by b number of places	20 >> 2	5

Tip: Bitwise operators are extremely fast. For example the left shift operator `<<` multiplies a number with 2. The speed of `a<<1` is greater than `a*2`.

Assignment Operators:

These operators assign the calculated value to a variable. So far we have seen `=` operator that assigns a value to the variable. Other assignment operators use combinations of other operators with `=` for a shorter syntax. Observe the operators below:

Operator	Syntax	Equivalent to
=	a = b	Assigns object b to a
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
//=	a //= b	a = a//b
**=	a **= b	a = a**b
%=	a %= b	a = a%b
&=	a &= b	a = a&b
|=	a |= b	a = a|b
^=	a ^= b	a = a^b
>>=	a >>= b	a = a>>b
<<=	a <<= b	a = a<<b

Note: Like other languages, python does not provide increment `a++` or decrement `a--` operators. Hence we have to use `a+=1` and `a-=1` instead.

Membership Operators:

Membership operators are used to identify whether an element is present in a collection or not.

Operator Name	Symbol	Description	Example	Result
In operator	**a in b**	Returns True if element a is present in collection b.	6 in [4, 5, 6, 7]	True
Not in operator	**a not in b**	Returns True if element a is not present in collection b.	6 not in [4, 5, 6, 7]	False

---

Identity Operators:

When we check the equality of two objects using `==` operator, python checks their value. But the identity operators `is` and `is not` compare the `id` of the objects.

Operator Name	Symbol	Description
Is operator	**a is b**	Returns True if element a and b refer same object.
is not operator	**a is not b**	Returns True if element a and b do not refer to same object.

We will eventually see these operators in our programs as we move further.

In this chapter, we have learned about operators available in python. So far we have been using data that is pre written in our programs. In the next chapter we will learn how to write user interactive programs, how to take user input and how to print data on console effectively.

Chapter 6: User Input and Output

Sometimes, the developer can not rely on hard coded values and he needs that a user must provide the value as an input to the program. This makes the program more usable and interactive. In Python, user can provide input from the console using `input()` function. `input()` is another inbuilt function of Python. In this chapter, we will explore `input()` and `print()` functions to make our programs responsive and interactive.

Taking Input from user:

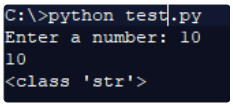
Function	**input([prompt])**
Description	Returns the user input as a string.
Parameters	*prompt* (optional) is an string that is displayed in console.

> Please note that the parameters enclosed in square brackets `[]` are optional.

Lets understand by an example. Create a new python file `hello.py` and add the following code:

```
1 n = input("Enter a number: ")
2 print(n)
3 print(type(n))
```

Now execute the above code. The console will display the message `Enter a number:` and execution will freeze. Enter any number and hit `enter`. You should see the following output:



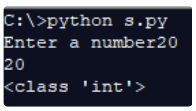
From above output, we can notice few things:

- `input()` function stops the execution until the user has not provided the input.
- The `prompt` string is displayed on the console.
- The input is assigned to the variable and it is by default a `str` object.

But what if we wanted to take an integer input? Python does not provide any implicit method for this. But we can typecast the input into `int` datatype easily. See the example below:

```
1 n = input("Enter a number: ")
2 n = int(n)
3 print(n)
4 print(type(n))
```

Output:

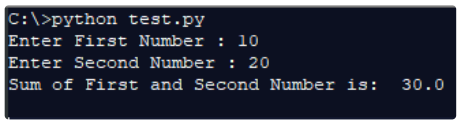


We can see that now our input is of `int` type. Similarly, we can cast the string to other types too.

Lets see one more example:

```
a = float(input("Enter First Number : "))
b = float(input("Enter Second Number : "))
c = a + b
print("Sum of First and Second Number is: ", c)
```

Output:



We have taken two inputs from user, converted them to `float` and printed their sum on the console.

Showing Output in console:

Function	**print(*obj [, sep=separator, end=end, file=file, flush=flush])**
Description	Returns the user input as a string.
Parameters	• **obj* represents one or more object/objects to be printed. They will be converted to string before printing on the console. • *sep*(optional) is the separator string. By default it is a single space `" "`. • end (optional) is the character to be printed at the end. By default it is the newline character `\n`. • *file* (optional) is the object having write() method. It can be a file or stream. By default it is `sys.stdout` that is our console. • *flush*(optional) is a boolean value which is `False` by default. If set to `True`, the stream will be flushed forcibly.

Lets see the `print()` function in details through some examples:

Printing a single object:

We can pass the object inside the print function directly. The object is converted to `str` before printing

```
# Printing a single object
print("This is a single string")
print(28)
```

Output:

```
This is a single string
28
```

Printing multiple objects:

We can pass multiple objects using comma `,` into `print()` function. By default they will be printed with a single space between each string.

```
# Printing multiple objects
print(1,2,3,4)
print("This", "is", 10, "line")
```

Output:

```
1 2 3 4
This is 10 line
```

Printing multiple objects with a separator:

By default we get a single space between the objects because `sep` is set to `' '`. We can use any other -character or string by passing it in `sep` argument. For example, we can write the data in a csv file using `,` in between the values.

```
# Printing multiple objects seperated with comma
print(1, "Two", 3.0, 'four', sep=', ')

# Printing multiple objects seperated with colon
print(1, "Two", 3.0, 'four', sep=':')
```

Output:

```
1, Two, 3.0, four
1:Two:3.0:four
```

Printing in same line with multiple print() functions:

All the `print()` calls we have seen, prints the object in a new line. This happens because the `end` parameter is by default `\n` and it prints a newline at the end. We can use any other character or string too.

```
# Printing two or more objects in same line
print("One", end=" ")
print("Two", end=" ")
print("Three")
```

Output:

```
One Two Three
```

Printing to a file:

The `file` parameter points to system console by default. We can print to any object that has a `write()` method available. We can also print to a file.

> In python, we can open a file using inbuilt `open()` function. We can open a file by passing its name and method. For example, `open("filename.txt", 'w')`, will open the file `filename.txt` in `write` mode. We will discuss this function in detail in upcoming chapters.

Lets see an example:

```
# Printing in a file
file_object = open("example.txt","w")
print("This is some dummy text", file=file_object)
```

Here, `file_object` is a file opened in `write` mode. The `print()` method will print the object in the file. When you execute the above code, a new file will be created and it will have the same content.

```
C:\>python test.py

C:\>dir

08/27/2023  03:58 PM                25 example.txt
08/27/2023  03:57 PM                93 test.py

C:\>type example.txt
This is some dummy text
```

Printing Sequences with single print() statement:

Any iterable or collection can be printed in single line with asterisk operator `*`. This operator spreads any iterable in the function. See the examples below:

Code Block

```
# Printing space separated list
l = [1,2,"3",4.0]
print(*l)

# Printing new line separated set
s = {"cat", "dog", "rabbit"}
print(*s, sep="\n")

```

Output:

Code Block

```
1 2 3 4.0
dog
rabbit
cat

```

In the above code, we have printed the list in single line using the default single space separator. And we have also printed the values of `set` using the newline character `\n`.

In this chapter, we have learned how to take input from user and how to print the objects or variables in console. We have explored the `print()` function and learned various ways to use it. In the next chapter, we will learn about conditional statements in python.

Chapter 7: Conditional Statements

Till now, we have seen that all of the lines, that we write in our python script, get executed. But there can be some situations when we just want to execute a group of lines (called block) for a certain condition. We have seen this in our real life too. When we visit an ATM and enter wrong PIN, we don't get to operate our account. On the other hand if we enter correct PIN, we get the options to operate the account. Similarly, if we try to withdraw more than a limited amount, the transaction is declined. Otherwise we get the money out of the machine. This all happens based on some conditions and only corresponding functionalities are executed.

In this chapter, we will see how to write conditional statements in Python and how they are different from other languages.

The if clause:

The `if` statement enables the user to execute a certain block of code on a particular condition. If the condition is `True`, the code is executed otherwise it is skipped.

Syntax:

```
if <condition>:
    <statement 1>
    <statement 2>
```

Example:

```
if 7>0:
    print("7 is greater than 0")

if 0>7:
    print("0 is greater than 7")
```

Output:

```
7 is greater than 0
```

In the above example we can see that `7>0` is a conditional statement as it returns either `True` or `False`. Since the condition is returning `True`, the next statement is executed and we get the above output. The next line has the condition `0>7` which returns `False` and we can see that the corresponding `print` statement doesn't get executed.

> Note: Not only in `if` clause, but all the control structures use 1-tab indentation. The sub-blocks use relatively 1 greater tab. A control statement ends with a colon `:` and that is where python expects a indented block of code. We will learn more about this by observing more examples in this chapter and upcoming chapters.

Importance of Indentation:

We can notice that in the previous example, the statement after `if` clause has a 1-tab indentation. Like other languages, python doesn't start a block of code inside curly braces `{ }`. It uses indentation to recognize the block of code. If we don't write a block of code with proper indentation, a syntactic error will be raised:

```
if 7>0:

```

```
2 print("7 is greater than 0")
```

Output:

```
1     print("7 is greater than 0")
2     ^
3 IndentationError: expected an indented block
```

Not only in `if` clause, but all the control structures use 1-tab indentation. The sub-blocks use relatively 1 greater tab. A control statement ends with a colon `:` and that is where python expects a indented block of code. We will learn more about this by observing more examples in this chapter and upcoming chapters.

The if-else clause:

We have seen that the `if` clause executes a statement if the condition is `True`. But what if we want to execute some other code if the condition was `False`? The `else` statement enables us to write a block of code when the given condition is `False`.

Syntax:

```
1 if <condition>:
2     <statement 1>
3     <statement 2>
4 else:
5     <statement 3>
6     <statement 4>
```

Example:

```
1 if 0>7:
2     print("0 is greater than 7")
3 else:
4     print("0 is not greater than 7")
```

Output:

```
1 0 is not greater than 7
```

In the above code, the condition is `False`, hence line 2 is not executed and the control moves to line 4. Note that we are using 1-tab indentation to declare a block for `else` clause.


Note that we are using 1-tab indentation to declare a block for `else` clause.

The if-elif-else clause:

We already know the use of `if` and `else`. But our code may have more than one condition. In such cases, we use `elif` clause to check for other conditions before the optional `else` block is executed. There can be more than 1 `elif` statements. This is also known as `if-elif` ladder. The control checks the conditions one by one and when any condition if found `True`, the corresponding block is executed and rest statements are skipped.

Syntax:

```
1 if <condition>:
2     <statement 1>
3     <statement 2>
```

```
elif <condition>:
    <statement 1>
    <statement 2>
else:
    <statement 3>
    <statement 4>
```

Example:

```
a = 5
if a>5:
    print("a is greater than 5")
elif a==5:
    print("a is equal to 5")
else:
    print("a is less than 5")
print("Finished")
```

Output:

```
a is equal to 5
Finished
```

In the above code, the condition at line 2 was `False`, so the corresponding block was skipped.

Next, condition in line 4 was `True`, hence the code in line 5 was executed. The rest statements are skipped, control comes out of ladder and line 8 is executed.

The Syntax of if-elif ladder:

To write an error free `if-elif` ladder, we need to follow some rules:

1. The ladder starts with a single `if` statement. There can be only one `if` statement in a ladder and the ladder ends as soon as another `if` statement is introduced.
2. After the `if` statement, there can be multiple `elif` statements. Each `elif` must contain a unique condition otherwise it will not be executed even if it is syntactically correct.
3. At last, there can be an optional `else` statement. Each `if` statement can end with a single `else` block and an `else` block without `if` is invalid.

See some examples:

```
number = 10

if number%2 == 1:
    print("number is odd")
elif number%5 == 0:
    print("number is divisible by 5")
elif number%2 == 0:
    print("number is even")
else:
    print("No condition was true")
```

Output:

```
number is divisible by 5
```

We can see that conditions in line 5 and 7 both are `True` but only the first statement gets executed and rest ladder is skipped.

```
s = "hello"

if 'h' in s:
    print("line 4")
if 'x' in s:
    print("line 6")
elif 'l' in s:
    print("line 8")
```

Output:

```
line 4
line 8
```

We can see that, line 4 and line 8 got executed. It is because the ladder starts from line 5 because there can be only 1 `if` statement in a ladder.

Operators used in conditional statements:

Any operator that returns a boolean value can be used in a *condition*. For example, below are some comparison operators that are commonly used with `if` statements:

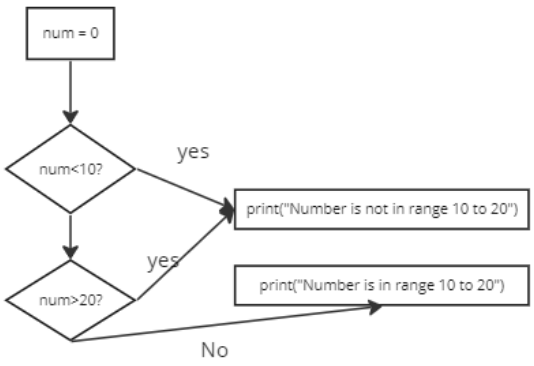
a == b (Equals)	a > b (greater than)	a < b (less than)
a != b (Not equals)	a >= b (greater than or equal to)	a <= b (less than or equal to)

Conjunction of multiple conditions:

We can combine more than one condition in a single line using `and` and `or` operators. For example, consider the code below:

Code Block

```
num = 0
if num < 10 or num > 20:
    print("Number is not in range 10 to 20")
else:
    print("Number is in range 10 to 20")

```



Output:

Code Block

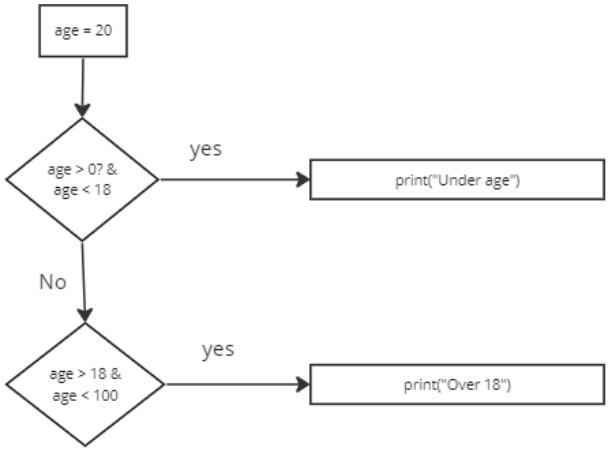
```
Number is not in range 10 to 20

```

We have used the `or` operator to combine two conditions. If either of the condition is `True`, it will return `True`. Similarly, if we use `and` operator, if either of the condition is `False`, the whole condition is `False`.

Code Block

languagepy

```
age = 20
if age > 0 and age < 18:
    print("Under age")
elif age > 18 and age < 100:
    print("Over 18")

```



Output:

Code Block

```
Over 18

```

We can see that in line 2, first condition `age > 0` is `True` but second condition after `and` operator `age<18` is `False`. Hence control moves to line 4 where both conditions are `True` and finally line 5 gets executed.

One liner control statements:

If our block contains only a single statement, we can write it in the same line. But this rule is not applicable for multi-line blocks.

```
a = 5
if a>5:print("a is greater than 5")
```

```
elif a==5:print("a is equal to 5")
else:print("a is less than 5")
```

Output:

```
a is equal to 5
```

Nested if-else statements:

We can write the conditional statements inside the other conditional statements. This kind of structure is called nesting. There is no limit of how deep the nesting can be.

Syntax:

```
if <condition>:
    if <condition>:
        ...
    else:
        ...
else:
    <statement>
    if <condition>:
        ...
    elif <condition>:
        ...
    else:
        ...
```

Example:

```
# Program to find greatest among 3 numbers
a, b, c = 10, 20, 30

if a>b:
    if a>c:
        print(a, "is greatest")
    else:
        print(c, "is greatest")
else:
    if b>c:
        print(b, "is greatest")
    else:
        print(c, "is greatest")
```

Output:

```
30 is greatest
```

Q: Is dangling else problem present in python?

A: No. Because python uses indentation to match the if and else. Hence the if block of any else block is found in the same indentation.

Switch Statement in Python:

Python does not provide `switch` keyword. But we can achieve the same functionality using two methods:

Method 1: Using if-else ladder:

This is the ideal method to implement `switch` statement because we can return the values as well as execute block of code inside a block. See the example below:

```
season = 3

if season == 0:
    print("Winter")
elif season == 1:
    print("Spring")
elif season == 2:
    print("Summer")
elif season == 3:
    print("Autumn")
else:
    print("Invalid season")
```

Output:

```
Autumn
```

Method 2: Using Dictionary:

We can use dictionary to return or print a particular value. But this method is not preferred because we can not execute a block of statement inside a dictionary.

```
season = 3

cases = {
    0: "Winter",
    1: "Spring",
    2: "Summer",
    3: "Autumn"
}

if season in cases:
    print(cases[season])
else:
    print("Invalid season")
```

Output:

```
Autumn
```

In the above code, we first checked whether the given case is present in the dictionary or not. Then we printed the corresponding value.

Ternary Operator:

In python, the ternary operator is not available with `? :` syntax. But there is a special syntax for ternary operation.

Syntax:

```
<Expression 1> if <Condition> else <Expression2>
```

Expression 1 is evaluated if the condition is `True`, otherwise Expression 2 is evaluated. Note that the `else` statement is mandatory here.

> Note that the `else` statement is mandatory here. We can not write the `if` alone.

Example:

```
# Program to find greatest among 2 numbers
a, b = 10, 20
c = a if a>b else b
print("Greatest number is ", c)
```

Output:

```
Greatest number is 20
```

The pass keyword:

Sometimes we don't know what to write in a certain condition but we want to write the other parts of code. It happens when we are not sure about the functionalities of the application. In such cases, Python provides a filler statement with `pass` keyword. Note that `pass` keyword is just a filler to suppress the syntactical errors.

```
l = [1, 2, 4]

if 3 not in l: # Not sure what to do
    pass
else:
    print("3 is present in list")
```

In the above code, nothing will be printed because we have not provided the implementation of `if` block.

In this chapter, we have learned how to write conditional statements and `if-elif-else` ladder. We have also learned about nested `if-else`, switch statement alternatives as well as a special syntax for ternary operator. In the next chapter, our focus will be on repetitive execution through loops. We will learn more about why and where to use loops in our python programs.

Chapter 8: Loops in Python

Loops are used for executing a set of statements for a fixed number of times. For example, if we want to display a table on console, we don't need to print each row individually. We can loop over the rows and it will drastically reduce the number of lines.

In this chapter, we will explore different type of loops with some extra features that are designed specifically for python.

The For loop:

In python, `for` loop is used to iterate over an iterable object like list, string etc. We use the `in` operator with `for` keyword to iterate over the iterables. An iterator variable is used as a reference to each of the element present in iterable.

Syntax:

```
for <iterator> in <iterable>:
    <statement 1>
    <statement 2>
    ...
```

Example:

```
animals_list = ["cat", "dog", "rabbit", "horse"]

for animal in animals_list:
    print(animal)
```

Output:

```
cat
dog
rabbit
horse
```

In the above code, we have iterated over the `animals_list` object. The iterator variable `animal` is used to refer each element of the list one by one. In this example we simply printed the elements on console. Note that just like `if-else`, we have used 1-tab indentation to write the block of code.

Lets see one more example:

Example:

```
l = [1,2,3,4,5,6]
result = 1
for num in l:
    result *= num
print("The multiplication is:",result)
```

Output:

```
The multiplication is: 720
```

We have simply multiplied all the numbers in the list using a `for` loop.

The range() function:

In other programming languages, we iterate over an array by using their index. `range()` is an inbuilt function of python that returns an iterable that represents a list of numbers. There are multiple ways to use a `range()` function.

Function Syntax	Description	Example
range(*N*)	Returns an iterable with integers from 0 to *N-1*.	range(5) = (0,1,2,3,4)
range(*start, end*)	Returns an iterable with integers from *start* to *end*-1.	range(2,5) = (2,3,4)
range(*start, end, diff*)	Returns an iterable with integers from *start* to *end*-1 with an interval of *diff*.	range(2,9,3)= (2, 5, 8)

Lets understand by an example:

Example:

```
countries = ["India", "USA", "China", "Japan",
             "Brazil", "Afganistan", "Nepal", "South Korea"]

print("Using First Syntax:", end=" ")
for i in range(6):
    print(countries[i], end=" ")
print()

print("Using Second Syntax:", end=" ")
for i in range(2, 7):
    print(countries[i], end=" ")
print()

print("Using Third Syntax:", end=" ")
for i in range(1, 8, 3):
    print(countries[i], end=" ")
print()
```

Output:

```
Using First Syntax: India USA China Japan Brazil Afganistan
Using Second Syntax: China Japan Brazil Afganistan Nepal
Using First Syntax: USA Brazil South Korea
```

The while loop:

A `while` loop is used when we want to loop with a condition. A `while` loop uses a condition with it, which is `True` for a fixed number of iterations and as soon as the condition becomes false, the loop breaks.

Syntax:

```
while <condition>:
    <statement 1>
    <statement 1>
    ...
```

Example:

```
num = 10
sum = 0
while num>0:
    sum += num
    num-=1
print("The sum is:", sum)
```

Output:

```
The sum is: 55
```

In the above code, we initialized the `num` variable. In the condition of `while` loop, we are checking the value of `num` and inside the loop we add it to the `sum` variable. As soon as `num` becomes 0, the loop ends and we get the above output.

The break statement:

A `break` statement is used to end the loop on some certain condition. When the interpreter encounters `break` statement, it stops the execution of its parent loop and the control moves to the next line after the end of that block.

Example:

```
for i in range(1,10):
    if i%4 == 0:
        break
    print(i, end=" ")
print("\nloop finished")
```

Output:

```
1 2 3
loop finished
```

In the above code, we iterate over the range 1 to 9. As soon as first integer that is divisible by 4 is encountered, the loop exits and the next line is printed. Note that we can nest `if-elif-else` inside the loop and vice-versa.

The continue statement:

`continue` is another statement that controls the flow of execution of a loop. When `continue` is encountered in the loop, the rest block of code is skipped and control moves to the start of the loop body. `continue` statement does not ends the loop but it stops the execution of the other statements.

Example:

```
for i in range(1,10):
    if i%4 == 0:
        continue
    print(i, end=" ")
print("\nloop finished")
```

Output:

```
1 2 3 5 6 7 9
```

```
2 loop finished
```

In the above code, we can see that while iterating over the range 1 to 9, the control could not reach to line 4 for `4` and `8` because the `continue` statement was executed in their case. Hence all numbers except `4` and `8` are printed.

---

else block with loops:

In python, there can be an optional `else` block too. The `else` block is executed when the loop ends without breaking in between. In other words, when the break statement executed in a loop, the `else` block will not get executed. This is one of the unique features of python.

Example:

```
1 list_of_odd_numbers = [1,3,5,7,9,11,13]
2
3 for number in list_of_odd_numbers:
4     if number%2 ==0:
5         break
6     print(number, end=" ")
7 else:
8     print("\nAll numbers are odd")
```

Output:

```
1 1 3 5 7 9 11 13
2 All numbers are odd
```

We can see that the `break` statement was not executed, hence the `else` block was executed. Lets see one more example:

Example:

```
1 num = 10
2 while num>0:
3     if num%4 == 0:
4         break
5     print(num, end=" ")
6     num-=1
7 else:
8     print("Not divisible by 4")
```

Output:

```
1 10 9
```

Since the loop abruptly exited, the `else` block was not executed.

Single statement in loops:

Just like `if-else`, if we have a single line block, we can write it in the same line instead of using the indented block. For example:

```
1 for i in range(5):print(i, end=" ")
```

Output:

```
1 0 1 2 3 4
```

Nested loops:

Loops can be nested inside other loops. Loops can also be nested inside conditional statements. The inner loop gets executed in each iteration of the outer loop. See the example below:

```
# Printing pairs with even sum
for x in range(5):
    for y in range(5):
        if (x+y)%2 == 0:
            print(x, y, end="|")
```

Output:

```
0 0|0 2|0 4|1 1|1 3|2 0|2 2|2 4|3 1|3 3|4 0|4 2|4 4|
```

---

pass keyword in loops:

We can use `pass` keyword as a filler in the loops too. Though it is a rare case to leave a loop empty but it can be used to avoid syntactical errors.

```
for i in range(10):
    pass
```

Looping in different data structures:

As we discussed before, we can iterate over any iterable data structure. Lets see how to iterate in various data structures with the help of an example:

Code Block

languagepy

```
# Iterating through list
cities = ["Kanpur", "Nagpur", "Pune", "Agra"]
for city in cities:
    print(city, end=" ")
print()

# Iterating over a string
string = "animation"
for character in string:
    print(character, end=" ")
print()
# Iterating over tuple
vowels = ('a', 'e', 'i', 'o', 'u')
for vowel in vowels:
    print(vowel, end=",")
print()

```

Output:

Code Block

```
Kanpur Nagpur Pune Agra
a n i m a t i o n
a,e,i,o,u,

```

We can also iterate over unordered data structures like `set` and `dict`.

Code Block

languagepy

```
# Iterating over a set
primes = {3, 2, 5, 11, 7}
for number in primes:
    print(number, end=",")
print()

# Iterating over a dict
primes = {3:'three', 2:'two', 5:'five', 11:'eleven', 7:'seven'}
for number in primes:
    print(number, primes[number], end=", ")

```

Output:

Code Block

```
2,3,5,7,11,
3 three, 2 two, 5 five, 11 eleven, 7 seven,

```

We can see that the elements are printed but the order is not same in case of `set`.

> **Note :** Since python 3.6, the dictionaries are able to maintain the order of insertion. It was previously possible only through *OrderedDict.*

In this chapter, we have learned about `for` and `while` loops. We have also learned about `break`, `continue`, `else` block, `range()` function and nesting in loops. In the next chapter, we will learn how to define our own functions and different ways to pass arguments to them.

Chapter 9: Functions in Python

In a large application, there are several functionalities. Each functionality can be used more than one time in different parts of the application. Instead of rewriting the code each time, we can create an individual part of code, which can be executed on demand. We achieve it with the help of functions.

A function is a set of code which is designed for a specific task. A function takes some input, processes it and returns an output. Functions are independent and reusable in nature. In this chapter, we will focus on how to write functions in python and how python functions are different from other languages.

Creating and calling a function:

To create a function, we use the `def` keyword. Just like the control statements, the function has a body that we write in 1-tab indentation. Below is the syntax of defining and calling a function:

Syntax:

```
# Declaring a function
def <function_name> (<parameters>) :
    <statement 1>
    <statement 2>
    ...
    return <value to return>

# Calling a function
<function_name>(<parameters>)
```

The function name follow that same naming convention we use to create a variable. The `return` keyword is used to return the value from function. When `return` is encountered, the control moves back to where the function was called. The inputs used by function (called arguments or parameters) are provided in round brackets `( )` while calling the function and they are also declared in function definition.

Example:

```
# Function to add two numbers

def add_numbers(a, b):
    c = a + b
    return c

# Calling the function

sum = add_numbers(10, 20)

print("Sum is :", sum)
```

Output:

```
Sum is : 30
```

We have defined a function `add_numbers` that takes two arguments, calculates their sum and returns it. We call the function in line 9 and capture the returned value in `sum` variable. We can see that we get the desired value.

> Note: The number of passed arguments must match the number of declared arguments.

Types of functions:

There are two types of functions:

1. Built-in Functions:

Built-in functions are predefined in the language. We can use them but we can not modify them. So far we have used several built-in functions like `print()`, `open()`, `input()`, `id()`, `type()` etc.

2. User defined Functions:

As the name suggests, these functions are written by the developer. Our focus will be on user defined functions throughout this chapter.

Return values of function:

In most of the programming languages, functions are bound to return a single data type. But thanks to dynamic typing, this is not the case in python. A developer can write a program that returns more than one kind of data type. See the code below:

```
# Function to divide two numbers

def divide(a, b):
    if b == 0:
        return "Division not possible"
    else:
        return a/b

result1 = divide(10,5)
result2 = divide(10,0)

print("Result 1 is :", result1)
print("Result 2 is :", result2)
```

Output:

```
Result 1 is : 2.0
Result 2 is : Division not possible
```

We can see that the function returned an integer in the first call and the same function returned a string in the next call.

If we don't return any value from a function, `None` is returned by default. A function can also return more than one value with the help of `tuple`. See the example below:

```
# Function to divide two numbers

def add_and_divide(a, b):
    if b != 0:
        sum = a+b
        div = a/b
        return sum,div

result1 = add_and_divide(10,5)
```

```
print("Return type is", type(result1))

result2 = add_and_divide(10,0)

print("Result 1 is :", result1)
print("Result 2 is :", result2)
```

Output:

```
Return type is <class 'tuple'>
Result 1 is : (15, 2.0)
Result 2 is : None
```

The above function returns two values when `b` is not 0. But we haven't defined what should happen if `b` is 0. In the first call, two values are returned as a tuple. See the output line 1 and 2. When we passed `b` as 0, nothing is returned, hence `None` was returned.

Arguments:

Arguments (or parameters) are the inputs provided to the function. In python, all variables point to a physical object in memory. Hence all the arguments are reference to real object. This is also know as "pass by reference". When we modify the object inside the function, it is also modified globally. See the example below:

```
l = [1,2,3]

def modify(l):
    # Modify the first element
    l[0] = 5

print("Before function call:", l)
modify(l)
print("After function call:", l)
```

Output:

```
Before function call: [1, 2, 3]
After function call: [5, 2, 3]
```

We can see that when we modified the list in function, it was globally modified.

Q: What is the difference between Arguments and Parameters?

A: Both words are used interchangeably. But the literal meaning is that 'arguments' are the values that we pass while calling the function and 'parameters' are the variables that we declare while defining the functions.

Types of Arguments:

In python, there are many ways we can pass arguments to a function. Lets see them one by one:

1. Positional Arguments:

These arguments are recognised by their position. If we change the position, we will get undesired results.

```
def print_vars(a,b,c):
    print("a is:", a)
    print("b is:", b)
    print("c is:", c)

a, b, c = "a", "b", "c"
print_vars(b, c, a)
```

Output:

```
a is: b
b is: c
c is: a
```

We can see that passed variables are mapped to corresponding variables in the same position.

2. Keyword Arguments:

If we don't want to remember the positions of arguments, we can use keyword arguments. When we call the function, we pass the values with argument names. Hence the order does not matter.

```
def print_vars(a,b,c):
    print("a is:", a)
    print("b is:", b)
    print("c is:", c)

A, B, C = "a", "b", "c"
print_vars(b=B, c=C, a=A)
```

Output:

```
a is: a
b is: b
c is: c
```

Notice how we passed the arguments. The order is shuffled but the values are mapped to the corresponding arguments.

3. Arbitrary Arguments:

Sometimes we want to pass an unknown number of arguments in our functions. In such cases we use arbitrary arguments. Arbitrary arguments are represented by a preceding `*` symbol, commonly used as `*args`. Such arguments are in the form of a tuple. We have seen this type of argument in our `print(*obj)` function where we pass any number of objects to print. The number of arguments can be 0 too.

```
def mutiplication(*nums):
    print(type(nums))
    result = 1
    for num in nums:
        result *= num
    return result

mul = mutiplication(4, 6, 3, 2, 1)
print("Multiplication is :",mul)
```

Output:

```
<class 'tuple'>
Multiplication is : 144
```

We can see that the argument `*nums` is of `tuple` data type. We can pass any number of arguments in our function.

> Note: There can be at most one arbitrary argument.

4. Arbitrary keyword arguments:

These arguments are unknown in number but each argument comes with its own name. They are represented by preceding `**`, commonly used as `**kwargs`. The arguments are passed in the similar syntax as keyword arguments and the `kwargs` variable is of `dict` datatype. The values can be accessed inside the function using the keys of the dictionary.

```
def print_languages(**kwargs):
    # Type of kwargs
    print(type(kwargs))

    # Lets see what is inside kwagrs
    print(kwargs)

    # Access the values from kwargs
    print("Language of USA is:", kwargs['usa'])

print_languages(china="chinese", usa="english", japan="japanese")
```

Output:

```
<class 'dict'>
{'china': 'chinese', 'usa': 'english', 'japan': 'japanese'}
Language of USA is: english
```

We can see that the passed arguments are automatically converted into a dictionary with string keys. We can use it inside the function just like a normal dictionary.

> Note: There can be at most one arbitrary keyword argument.

5. Default valued arguments:

Default arguments have an initial value provided with `=` operator while declaring them. If the user provides the value, it is used, otherwise the default value is used. We have seen that in `print()` function, the arguments like `end`, `sep` have some default values and they change only when we provide it. See the example for clarification:

Code Block

languagepy

```
def myplanet(planet = "Earth"):
    print("I live on", planet)

# Using default value
myplanet()
# Using other value
myplanet("Mars")

```

Output:

Code Block

```
I live on Earth
I live on Mars

```

> Note that default arguments become optional by default. We can omit them if we want to use the default value itself. As a good practice, always use keyword while passing value to default argument.

Ordering of the arguments:

Since python provides a variety of arguments, we must follow some rules if we want to use them in the same function. Python interpreter uses a fixed order to understand which argument should be mapped to which:

...

1. First all the positional arguments must be declared.
2. If there is any arbitrary argument, it should be declared.
3. Then all the default arguments should be declared. They should be used only with their keyword in such cases.
4. At last, the arbitrary keyword argument is declared.

Example:

Code Block

languagepy

```
def example_function(a, b, *args, c=10, d=20, **kwargs):
    print("postional arguments:", a, b)
    print("default arguments:", c, d)
    print("arbitrary arguments:", args)
    print("arbitrary keyword arguments:", kwargs)

example_function(2,5,10,60, c=8, e=70)

```

Output:

Code Block

```
postional arguments: 2 5
default arguments: 8 20
arbitrary arguments: (10, 60)
arbitrary keyword arguments: {'e': 70}

```

You can observe how the arguments are mapped.

> Note: When we use both *args and default arguments, make sure to use default arguments by their keyword. Otherwise they will be mapped to *args.
>
> If we don't follow this order, the code will raise syntactical errors.

...

The pass keyword in functions:

Just like the conditional statements, if we are not sure what to do inside a function, we can write `pass` keyword as a filler to avoid syntactical errors.

Code Block

languagepy

```
def example():
    pass

```

...

Recursive Functions in Python:

Python supports recursive functions too. A recursive function calls itself inside its own body. When we write a recursive function, we have to include a base condition that defines where to stop calling the function.

Need of recursion:

Recursion is needed when we have a problem that can be divided into subproblems. Doing this drastically reduces the lines of code used and increases the readability of program.

Many problems related to data structures and algorithm are solved in few lines of code with the help of recursion. Recursion is the backbone of all the interview topics like Trees, Graphs, Dynamic programming and Backtracking. Hence it is important to understand how recursion works in python.

> Note: We repeatedly call the function from from the function itself. Hence we must be careful while writing a recursive program. If we don't write a base condition properly, the program will go into an infinite loop.
>
> The best practice is that we should check the base condition first and if it is `True`, we should return immediately. After that we should write the logic. Make sure that the problem is getting smaller in each call because sometimes we do the opposite and the program is messed.

Lets see how we can calculate factorial of a number using recursive function:

```
# Program to find factorial of a number
# factorial of n is equal to n time factorial of n# Factorial of n is equal to n times factorial of n-1
# And factorial of 1 is 1 itself. This is the base condition
# Hence factorial of 3 is 3*2*1 = 6# Hence factorial of n is n*(n-1)*(n-2)*...2*1

def my_factorial(n):
    # Base condition
```

```
    if n <=1:
        return 1

    return n * my_factorial(n-1)

print("Factorial of 5: ", my_factorial(5))
```

Output:

Code Block

```
Factorial of 5:  120

```

> Although recursion is available in Python, it is not recommended to use recursive functions. Many python experts have reported that recursion is too slow and shouldn't be used if we already don't know how much depth is required. Python is not designed to use recursion and it has an limited inbuilt depth that we have to increase manually. Hence, whenever possible, we should try to convert our recursive functions into iterative ones.

...

The doc string:

Python provides easy documentation. We can provide the details of our function in the very first line of the function body in the string format. That string can be accessed using `__doc__` property of the function.

Code Block

languagepy

```
def print_table(n):
    "This function prints the table of the given number n"
    for i in range(1, 10):
        print(n*i, end=" ")

print(print_table.__doc__)
print_table(7)

```

Output:

Code Block

```
This function prints the table of the given number n
7 14 21 28 35 42 49 56 63

```

Notice how we got the string description through the `__doc__` property.

> The developers use the doc string extensively to provide documentation. It saves a lot of time because they are easily captured by modern IDEs and we don't need to waste our time on extra documentation.

...

In this chapter, we declared our own functions and learned how to use them. We also learned how python function can return more than one type of result and the different types of arguments python provides.

In the upcoming chapters, we will explore the list data structure of python. We will see how to use list and what are the different parameters provided by the inbuilt list.

Chapter 10: Python List

What are collections:

When we work with a small set of variables, we declare them individually in the program. However, even in simple programs, we may need to work with thousands or even millions of objects. To store them, creating variables for each is not a good idea. In such cases, we need to use collections that can hold millions of objects.

Choosing the right collection data structure:

We can choose a collection based on our requirements:

1. When we need a collection that stores data sequentially, can store duplicate objects and can be modified easily, we use `list`.
2. However if we don't want to modify the data but want only to read it, `tuple` is the best choice. `tuple` is also ordered and can store duplicate objects.
3. When we are working with unique objects, we should use `set`. A `set` does not allow duplicates and it is unordered. But we can search the objects very quickly in a `set`.
4. When we want to use or modify objects through a `key`, we use `dict`. A `dict` contains key-value pairs and it can be accessed very quickly. However the keys are unique for a one to one mapping.

In this chapter we will learn the `list` data structure in detail along with its methods.

List is one of the most powerful data structures in Python. A `list` object can hold an ordered collection of heterogeneous objects. Here "ordered" means that each object has its own position in list and "heterogenous" means that the objects need not to be of same data type or same class.

Initialization of list:

We can initialize a list using empty square brackets `[ ]` as well as `list()` constructor.

Code Block

languagepy

```
listObj = []
listObj = list()

```

We can initialise a list with some objects by separating them using commas.

Code Block

languagepy

```
# List of numbers
l1 = [1,2,4]
# List of Strings
l2 = ['hello', 'world']
# List of heterogeneous objects
l3 = [1, 'hello', {'key':'value'}, 8.90, 6+9j]
```

Accessing Data from list:

1. We can directly point to the objects in list using their indices. List is Mutable, so we can also

modify the items in list using their indices.

Code Block

languagepy

```
# Initial list
vowels = ['a', 'e', 'i', 'o', 'k']
print("At line 3:", vowels[0], vowels[3])
vowels[4] = 'u'
print("Now vowels is", vowels)
print("At line 6:", vowels[4])

```

Output:

Code Block

```
At line 3: a o
Now vowels is ['a', 'e', 'i', 'o', 'u']
At line 6: u

```

But we cannot access the indices that are not present

Code Block

languagepy

```
vowels = ['a', 'e', 'i', 'o', 'k']
print(vowels[6])

```

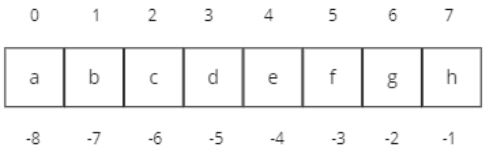
Output:

Code Block

```
IndexError: list index out of range

```

1. List supports negative indices. The

`A[-X]` index is similar to `A[len(list) – X]`. Hence we can access
indices up to length of list.



Code Block

```
# We can access negative index in list
vowels = ['a', 'e', 'i', 'o', 'k']
print(vowels[-2])

```

Output:

Code Block

```
o

```

> Note: `len()` is an inbuilt function that returns the number of items in a collection. It can be applied on any iterable as well as any object that has `__len__()` method implemented.

However, we can not access the element that is not inside the range.

Code Block

languagepy

```
# We can not access the indices that are unavailable
vowels = ['a', 'e', 'i', 'o', 'k']
print(vowels[-6])

```

Output:

Code Block

```
IndexError: list index out of range

```

1. We can access a particular part of the list using `:` separator.

Code Block

languagepy

```
vowels = ['a', 'e', 'i', 'o', 'k']
# Print from index 1 to 2
print(vowels[1:3])
```

```
4
5 # Print from index 2 to last
6 print(vowels[2:])
7
8 # Print from start to 2nd index
9 print(vowels[:3])
10
```

Output:

Code Block

```
1 ['e', 'i']
2 ['i', 'o', 'k']
3 ['a', 'e', 'i']
4
```

Methods of list:

List supports many methods to programmatically manipulate it. Some of the commonly used methods are being discussed here.

> Note that the arguments in square brackets [ ] represents optional parameters.

Method Name	Description
append(item)	Appends the item at end
pop([index])	Removes the item from end or given index
insert(index, item)	Inserts the item at specified index
reverse()	Reverses the list
remove(item)	Removes the specified item
copy()	Returns a shallow copy of list
clear()	Clears the list
sort([key, reverse])	Sorts the list

Lets understand each method in detail with the help of examples:

1. append(item) :

This method adds the given item to last of list and increases list size by 1.

```
1 # Initialize the list
2 l = [1, 2, 3]
3
4 # Add 4 to the list
5 l.append(4)
6 print("Line 6:", l)
7
```

```
# Add 'python' to the list
l.append('python')
print("Line 10:", l)
```

Output:

```
Line 6: [1, 2, 3, 4]
Line 10: [1, 2, 3, 4, 'python']
```

2. pop([index]):

If index is provided, it will remove and return the item present at that index.
Otherwise the last item of list is removed and returned. If the list is empty or the specified index is not valid, `IndexError` is raised.

```
fruits = ["apple", "banana", "grapes", "papaya"]
# Pop the last item
fruit = fruits.pop()
print("Popped Item:", fruit)
print("List :", fruits)

vowels = ['a', 'b', 'e', 'i', 'o', 'u']
# Pop the item at index 1
consonent = vowels.pop(1)
print("consonent is:", consonent)
print("List:", vowels)
```

Output:

```
Popped Item: papaya
List : ['apple', 'banana', 'grapes']
consonent is: b
List: ['a', 'e', 'i', 'o', 'u']
```

3. insert(index, item) :

Inserts the item to the specified index. Shifts the rest list to the right by 1 position.

```
years = [2017, 2019, 2020, 2021]
# Insert 2018 at index 1
years.insert(1, 2018)
print(years)
```

Output:

```
[2017, 2018, 2019, 2020, 2021]
```

4. reverse():

This method reverses the list inplace. “Inplace” means that the list is modified instead of returning a copy.

```
letters = ["a", "b", "c", "d"]
letters.reverse()
print(letters)
```

Output:

```
['d', 'c', 'b', 'a']
```

5. remove(item):

This method searches the item from left and removes the first occurrence only. If the item is not found, a `ValueError` is raised.

```
animals = ['dog', 'cat', 'horse', 'cat']

# Remove cat from the list
print("Removing cat")
animals.remove('cat')
print(animals)

# Remove rabbit from the list
print("Removing Rabbit")
animals.remove('rabbit')
print(animals)
```

Output:

```
Removing cat
['dog', 'horse', 'cat']

Removing Rabbit
ValueError: list.remove(x): x not in list
```

We can see that when we removed “cat”, it was removed successfully. But removing “rabbit” was not possible and an error was raised.

6. copy():

This method returns a shallow copy of the list. Shallow copy means the items are copied but their internal data still refers to the same memory location. If we modify that data, it is reflected in the copy too.

```
l1 = [1, 2, 'hello', {'world':9+8j}]
l2 = l1.copy()
l3 = l1[:]
print(id(l1), id(l2), id(l3))
print(l1)
print(l2)
print(l3)
```

Output:

```
140039313228480 140039313228928 140039313289792
[1, 2, 'hello', {'world': (9+8j)}]
[1, 2, 'hello', {'world': (9+8j)}]
[1, 2, 'hello', {'world': (9+8j)}]
```

We can see that all lists contain same data but they are referring to different objects.

7. clear() :

This method clears the list ie deletes all the items of the list. It is same as setting `list=[]`.

```
l1 = [1, 2, 'hello', {'world':9+8j}]
l1.clear()
print(l1)
```

Output:

```
[]
```

8. sort([key, reverse]):

This method sorts the list in ascending order of the natural values if no parameter is passed. Numbers are sorted according to their values. Strings are sorted according to their lexicographical order.

```
l = [1,4,3,4,2,8]
# Sort in ascending order
l.sort()
print("Sorted list:", l)
```

Output:

```
Sorted list: [1, 2, 3, 4, 4, 8]
```

> Note: We can not sort heterogeneous list as two objects of different data types can not be compared. If we try to do so, we will get a `TypeError`.

If parameter `reverse=True` is set, it will sort the list in descending order. The argument reverse is `False` by default.

```
animals = ['dog','rabbit', 'cat', 'horse']
# Sort in descending lexicographic order
animals.sort(reverse=True)
print(animals)
```

Output:

```
['rabbit', 'horse', 'dog', 'cat']
```

If we want to sort according to a particular property of the object, we must provide a key function. The key is a function that takes the object and returns a number or string that is the comparable value of the object.

```
animals = ['dog','rabbit', 'cat', 'horse']
# Sort the array according to string length
animals.sort(key=len)
print(animals)
```

Output:

```
['dog', 'cat', 'horse', 'rabbit']
```

> Note that we only need to pass function name, we don't need to call it. Internally, python will use this function to compare the objects.

If two or more objects return same value in key and we want to resolve the conflict using some other property, function must return it as second element in a `tuple`.

```
def comparator(item:str):
    # Sort according to length. If length is same, sort by lexicographic order.
    return (len(item), item)

animals = ['dog','rabbit', 'cat', 'horse']
```

```
animals.sort(key=comparator)
print(animals)
```

Output:

```
['cat', 'dog', 'horse', 'rabbit']
```

In this chapter, we have learned all the useful stuff related to list. We learned different ways to access list and different methods to modify its content. We can now understand why list is called one of the most powerful data structures of python. In the upcoming chapter, we will learn about the `set` data structure and its methods.

Chapter 11: Python Set

When we need to store a large collection of unique items, `set` data structure is the best choice. A `set` stores only unique values and it is unordered. By unique values, we mean that a value can be inserted only once. Unordered means, the insertion order is not maintained as well as we can not access an item using its index. Like `list`, we can store heterogenous objects in a set. The values in a `set` can be created and removed but they can not be modified. In this chapter, we will learn about `set` and its methods.

When we need to store a large collection of unique items, `set` data structure is the best choice. It is one of the 4 collections available in python. Others are List, Dictionary and Tuple which are used in different scenarios. A `set` stores only unique values and it is unordered. By unique values, we mean that a value can be inserted only once. Unordered means, the insertion order is not maintained as well as we can not access an item using its index. Like `list`, we can store heterogenous objects in a set. The values in a `set` can be created and removed but they can not be modified. In this chapter, we will learn about `set` and its methods.

Initialization of set:

We can initialize a set by below methods:

A set `set()` with initial values can be declared with values inside curly braces `{ }`.

```
s = {1,2,3,3}
r = {'this', 'is', 0, 2.5}
print(type(s), s)
print(type(r), r)
```

Output:

```
<class 'set'> {1, 2, 3}
<class 'set'> {0, 'this', 2.5, 'is'}
```

> Note that `{ }` represents `dict` by default. This is why an empty set is also printed as `set()`.
>
> Notice how the duplicate `3` is removed from the set automatically. Also we can store heterogeneous data in a set.

1. A set `{ }` can be created from an existing iterable by passing it into the `set()` constructor.

```
countries = ['India', 'China', 'isJapan', 'India']
set_of_countries = set(countries)
print(set_of_countries)
```

Output:

Code Block

```
<class {'set'> {1, 2, 3}
<class 'set'> {0Japan', 'China', 'this', 2.5, 'is'}
```

1. A set can be created from an existing iterable by passing it into the `set()` constructor.

```
countries = ['India', 'China', 'Japan', 'India']
set_of_countries = set(countries)
```

```
print(set_of_countries)
```

Output:

```
{'Japan', 'China', 'India'}
```

---

1. An empty set can be initialized using

`set()` constructor.

Code Block

```
s = set()
print(type(s), s)
```

Output:

Code Block

```
<class 'set'>  set()
```

> Note that `{ }` represents `dict` by default. This is why an empty set is also printed as `set()`.

Accessing Data from a set:

As discussed before, we can not access the items present in set using their indices. However, we can check their availability or membership using `in` operator.

```
animals = {"zebra", "lion", "tiger"}

print("Rabbit is present?", "rabbit" in animals)
print("Tiger is present?", "tiger" in animals)
```

Output:

```
Rabbit is present? False
Tiger is present? True
```

Similarly, we can loop through the set using `in` operator with `for` loop. However, the order of items is not guaranteed to be same as the insertion order.

```
animals = {"zebra", "lion", "tiger"}

for animal in animals:
    print(animal, end=" ")
```

Output:

```
zebra lion tiger
```

> Q: Which is faster, `in` operator with `list` or `set`?

> A: `set` does not maintain any order. This is because it uses hashing internally. Hence searching any value becomes very fast. While in `list`, using `in` operator will search for the element linearly and it will take longer time as compared to `set`.

Length or Size of Set:

The inbuilt `len()` function is used to find the number of items present in a `set`.

```
s = {1,2,3}
print("Size of set is :", len(s))
```

Output:

```
Size of set is : 3
```

Methods of Set:

Set supports some methods that makes it easier to manipulate it and use it for various real life problems. We will discuss all the popular and most used method in detail one by one.

Method Name	Description
add(x)	Adds x to the set.
update(*iterable)	Adds the values from all the iterables.
discard(x)	Removes x from the set.
remove(x)	Removes x from the set. If x is not present, a `KeyError` is raised.
pop()	Removes a random value from the set.
clear()	Clears the set.
issuperset(iterable)	Returns `True` if the set is superset of given iterable.
issubset(iterable)	Returns `True` if the set is subset of given iterable.
isdisjoint(iterable)	Returns `True` if the set is disjoint of given iterable.
difference(*iterable)	Returns the set difference.
intersection(*iterable)	Returns the set intersection.
symmetric_difference(iterable)	Returns the symmetric difference.
union(*iterable)	Returns the set union.
copy()	Returns a copy of the set.

Lets see these methods with some examples.

1. add(x):

This method adds x to the set if it is not already present. If x is already there, nothing is added.

```
animals = {'cow', 'goat'}
animals.add('horse')
animals.add('cow')
print(animals)
```

Output:

```
{'cow', 'goat', 'horse'}
```

2. update(*iterable):

This method updates the set using the provided iterables. All the items in all the iterables are added if they are not present already.

```
primes = {2, 3, 5}

list_1 = [2, 3, 5, 7, 11]
list_2 = [2, 7, 13, 19]

primes.update(list_1, list_2)
print(primes)
```

Output:

```
{2, 3, 5, 7, 11, 13, 19}
```

3. discard(x):

This method will remove x from the set if it is present. Otherwise nothing will happen.

```
shopping = {"milk", "eggs", "butter"}
shopping.discard("bread")
shopping.discard("eggs")
print(shopping)
```

Output:

```
{'butter', 'milk'}
```

4. remove(x):

This method will remove x only if x is present in the set. However, if x is not present, it will raise a `KeyError`.

```
shopping = {"milk", "eggs", "butter"}
shopping.remove("eggs")
print(shopping)
```

Output:

```
{'butter', 'milk'}
```

5. pop():

This method will randomly returns and remove any of the element from the set. If there are no elements, a `KeyError` is raised.

```
squares = {1, 4, 9, 16}
popped = squares.pop()
```

```
print("Popped:", popped, "Remaining set:", squares)
```

Output:

```
Popped: 16 Remaining set: {1, 4, 9}
```

6. clear():

This method will clear the set.

```
numbers = {4, 7, 8, 23}
numbers.clear()
print(numbers)
```

Output:

```
set()
```

7. issuperset(iterable):

This method will return `True` if this `set` is the superset of elements present in iterable.

```
birds = {'parrot', 'nightangle', 'owl'}
more_birds = ['parrot', 'nightangle']
print(birds.issuperset(more_birds))
```

Output:

```
True
```

8. issubset(iterable):

This method will return `True` if this `set` is the subset of elements present in iterable.

```
birds = {'parrot', 'nightangle'}
more_birds = ['parrot', 'nightangle', 'owl']
print(birds.issubset(more_birds))
```

Output:

```
True
```

9. isdisjoint(iterable):

This method returns `True` when there is not even a single common element from this set and the provided iterable.

```
primes = {2,3,7}
more_primes = {11, 19, 13}

print(primes.isdisjoint(more_primes))
```

Output:

```
True
```

10. union(*iterable):

This method returns the union of this set and the iterables provided.

```
primes = {2,3,7}
more_primes = {7, 19, 13}
some_more_primes = {23, 29}
all_primes = primes.union(more_primes, some_more_primes)
print(all_primes)
```

Output:

```
{2, 3, 19, 29, 23, 7, 13}
```

11. intersection(*iterable):

This method returns the intersection of this set and the provided iterables.

```
animals1 = {"cow", "dog", "rabbit"}
animals2 = {"horse", "dog", "tiger"}
animals3 = ["elephant", "tiger", "lion"]
print(animals1.intersection(animals2, animals3))
```

Output:

```
set()
```

12. difference(*iterable):

This method returns the set difference of this set and the provided iterables.

```
animals_1 = {"cow", "dog", "rabbit"}
animals_2 = {"horse", "dog", "tiger"}
print(animals_1.difference(animals_2))
```

Output:

```
{'rabbit', 'cow'}
```

13. symmetric_difference(iterable):

This method returns the symmetric difference of this set and the provided iterable.

```
start = {1, 2, 3, 4, 5}
end = [4, 5, 6, 7, 8]
print(start.symmetric_difference(end))
```

Output:

```
{1, 2, 3, 6, 7, 8}
```

14. copy():

This method returns a copy of this set.

```
s1 = {1,2,4}
s2 = s1.copy()
print(s1, s2)
print(id(s1), id(s2))
```

Output:

```
{1, 2, 4} {1, 2, 4}
2815087319968 2815087319744
```

Operators of Set:

`set` data structure also supports some operators. However these operators internally use the methods discussed above. Hence we can quickly understand the working of these operators.

Operator	Description
A | B	Returns the union of set A and set B.
A <= B	Returns `True` if A is a subset of B.
A < B	Returns `True` if A is a proper subset of B.
A >= B	Returns `True` if A is a superset of B.
A > B	Returns `True` if A is a proper superset of B.
A - B	Returns the set difference of A and B.
A & B	Returns the intersection of A and B.
A ^ B	Returns the symmetric difference of A and B.

Lets see some examples using the operators:

- The `|` and `&` operator are used between two sets to find the union and intersection respectively.

Code Block

```
set1 = {2, 3, 4}
set2 = {3, 4, 6, 7, 8}
print("Union:", set1 | set2)
print("Intersection:", set1 & set2)

```

Output:

Code Block

```
Union: {2, 3, 4, 6, 7, 8}
Intersection: {3, 4}
```

Code Block

```
natural_numbers = {1, 2, 3, 4, 5}
integers = {0, 1, 2, 3, 4, 5}

print("natural numbers are a subset of integers :", natural_numbers<=integers)
print("integers are a superset of natural numbers :", integers>=natural_numbers)

```

Output:

Code Block

```
natural numbers are a subset of integers : True
integers are a superset of natural numbers : True

```

- The `<` and `>` operator are used between two sets to determine the proper subset and proper superset respectively.

Code Block

```
zoo = {'zebra', 'hippo', 'tiger'}
jungle = {'zebra', 'hippo', 'tiger'}

print('zoo is a proper subset of jungle:', zoo<jungle)

```

Output:

Code Block

```
zoo is a proper subset of jungle: False

```

- The `-` and `^` operators are used between two sets to find the set difference and symmetric difference respectively.

Code Block

```
shopping_list = {'toothpaste', 'milk', 'butter', 'eggs'}
bought = {'milk', 'butter'}
# Set Difference
print("Remaining Items: ", shopping_list - bought)

```

Output:

Code Block

```
Remaining Items:  {'eggs', 'toothpaste'}

```

Code Block

```
imports = {'electronics', 'food', 'machinery', 'vehicles'}
exports = {'food', 'electronics', 'cotton'}
# Symmetric Difference
print("Unique import and exports", imports^exports)

```

Output:

Code Block

```
Unique import and exports {'vehicles', 'machinery', 'cotton'}

```

...

Now we can see that how a `set` is different from other collections in python.

- A `list` contains ordered elements and allows duplicates. It is mutable.
- A `set` contains unordered elements and doesn't allow duplicates. It is immutable.
- A `dict` contains ordered elements from python 3.7 and its keys are non-duplicate.
- A `tuple` contains ordered items and allows duplicates. But it is immutable.

In this chapter, we learned what is `set` and what are the methods available for `set`. We learned how to manipulate the set by adding and removing items. We also learned how to perform the set operations using the methods as well as the operators. In the upcoming chapter, we will learn about `str` in detail.

Chapter 12: Python String

In programming, a string is a sequence of characters that represents some textual information. In python, we can use strings using `str` class. Python strings are immutable but fast. By immutable, we mean that python strings can be created and accessed, but we can not modify them. In this chapter, we will explore the `str` class and the methods available for strings.

Initialization of Strings:

A string can be initialized in multiple ways:

1. We can enclose the text inside single quotes

`' '` or double quotes `" "` to declare a string literal.

```
s = "This is a string"
r = 'This is another string'
```

1. We can declare multiline strings using triple quotation marks as demonstrated below:

```
s = ''' This is a
multiline string
demonstration'''

r = """ This is another
multiline string
demonstration """
```

1. We can also pass an object inside

`str()` constructor in order to create a string.

```
# Converting number to string
num = 567123
snum = str(num)
print(type(snum), snum)

# Converting list to string
l = [100,101,102]
s = str(l)
print(type(s), s)
```

Output:

```
<class 'str'> 567123
<class 'str'> [100, 101, 102]
```

> Q: How does `str()` constructor work?
>
> A: Each class that implements the `__str__()` method, can be used in `str()` constructor. This method returns the string representation of the object.

Accessing String:

A string is a series of character. Hence it is also an iterable. We can access the characters in string using the indices just like we do in case of `list`. See some examples below:

1. We can access the characters using their indices:

```
s = "Python Program"
print(s[0])
print(s[8])
```

Output:

```
p
r
```

1. Strings also support negative indices:

```
s = "Python Program"
print(s[-1])
print(s[-11])
```

Output:

```
m
h
```

1. We can access a part of the string using the

`:` operator in the following way:

```
s = "Python Program"
# s[m:]  --> from m-th index to end
print("First :", s[5:])
# s[:n]  --> from 0-th index to (n-1)-th index
print("Second :", s[:8])
# s[m:n] --> from m-th index to (n-1)-th index
print("Third :", s[3:11])
```

Output:

```
First : n Program
Second : Python P
Third : hon Prog
```

1. We can access characters at a particular distance using a second

`:` operator.

```
s = "0123456789"
# Print every 3rd character from 2 to 8.
print(s[2:9:3])
```

Output:

```
258
```

1. We can access the string but we can not modify it. It will raise a

`TypeError`.

```
s = "Batman"
s[0] = "C"
```

Output:

```
TypeError: 'str' object does not support item assignment
```

Size of String:

The `len()` function is used to get the size of the string.

```
s = "012345"
print(len(s))
```

Output:

```
6
```

> Q: How does the `len()` function work?
>
> A: The class that implements `__len__()` method can use the `len()` function. The classes like `list`, `set`, `str`, `dict` etc. already implement the `__len__()` method. We will talk about such methods in a later chapter.

Looping in String:

We can normally loop through a string just like we did in list.

1. Using

`for` loop:

```
s = "flower"
for i in s:
    print(i, end=",")
```

Output:

```
f,l,o,w,e,r,
```

1. Using

`while` loop:

```
s = "flower"
i = 0
while i<len(s):
    print(s[i], end=",")
    i+=1
```

Output:

```
f,l,o,w,e,r,
```

Substring Search:

We can check whether a substring exists in the given substring or not using `in` operator:

```
print("Python" in "This is a Python Program")
```

Output:

```
True
```

String Methods:

There are a lot of methods available for `str`

Method	Description
center()	Returns a center aligned string.
count()	Returns frequency of occurrence of specified substring.
encode()	Returns an encoded string.
endswith()	Returns `True` if the string ends with given substring.
find()	Returns index of specified substring, if present.
format()	Formats the string by using the specified description.
index()	Returns index of specified substring, if present.
isalnum()	Returns `True` if the string contains only alphanumeric characters.
isalpha()	Returns `True` if the string contains only alpha characters.
isdecimal()	Returns `True` if the string contains only decimal characters.
isdigit()	Returns `True` if the string contains only digits.
islower()	Returns `True` if the string contains only lowercase characters.
istitle()	Returns `True` if the first character of each word is in uppercase.
isupper()	Returns `True` if the string contains only uppercase characters.
join()	Joins and returns the string only iterable with the given string.
ljust()	Returns a left justified string.
lower()	Returns a lowercase string.
lstrip()	Returns the string after removing left blank characters.
partition()	Partitions the string about the specified substring.
replace()	Returns a string after replacing the specified source to target.
rfind()	Similar to find(). Searches from right end instead of left end.
rindex()	Similar to index(). Searches from right end instead of left end.

rjust()	Returns a right justified string
rpartition()	Similar to partition(). Searches from right end instead of left end.
rsplit()	Similar to split(). Searches from right end instead of left end.
rstrip()	Returns the string after removing right blank characters.
split()	Splits the string at given substrings and returns a list.
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Trims the left and right blank characters.
swapcase()	Converts uppercase to lower and vice versa.
title()	Converts the first character of each word to upper case.
translate()	Translates each match using the specified dictionary.
upper()	Converts all the characters into uppercase.

> A point worth notice is that all the methods return a new string. There is no modification that can be done and hence, each time a new `str` object is created. This is why string manipulation is a costly process.
>
> To avoid mistakes in programming, always remember that methods of immutable data structures will always return a new copy. So we must always assign the new string to a variable otherwise we may run into logical errors.

Formatting Strings:

We can use the following methods to format the string:

lower() & upper():

We can convert the given string into uppercase using `upper()` method and lowercase using `lower()` method.

```
alphabet = "abcdEFGhiJKlmNoPqrsTuvWxyz"
print("Lowercase:", alphabet.lower())
print("Uppercase:", alphabet.upper())
```

Output:

```
Lowercase: abcdefghijklmnopqrstuvwxyz
Uppercase: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

title():

This method converts the first character of each word to uppercase.

```
greeting = "good morning! have a good day!"
print(greeting.title())
```

Output:

```
Good Morning! Have A Good Day!
```

swapcase():

This method swaps the case of all characters from upper to lower and vice versa.

```
alphabet = "abcdEFGhiJKlmNoPqrsTuvWxyz"
print(alphabet.swapcase())
```

Output:

```
ABCDefgHIjkLMnOpQRStUVwXYZ
```

center(), ljust() & rjust():

To align a string to center, left and right, we use center(), ljust() and rjust(). We must pass the width of the string and a optional character to fill the space. By default empty space will be filled.

```
heading = "Heading"
print("Center align with space\t\t", heading.center(15))
print("Center align with character\t", heading.center(15, '*'))

print("Left align with space\t\t", heading.ljust(15))
print("Left align with character\t", heading.ljust(15, '-'))

print("Right align with space\t\t", heading.rjust(15))
print("Right align with character\t", heading.rjust(15, '#'))
```

Output:

```
Center align with space             Heading
Center align with character      ****Heading****
Left align with space            Heading
Left align with character        Heading--------
Right align with space                  Heading
Right align with character       ########Heading
```

strip(), lstrip() & rstrip():

We can trim the extra space, tabs and newline characters from the ends of the string. `lstrip()` will trim from left end, `rstrip()` from right end and `strip()` will trim from both ends.

```
string = "  Toy Story \t"
print("Without strip:","|"+string+"|")
print("Left Strip", "|"+string.lstrip()+"|")
print("Right Strip", "|"+string.rstrip()+"|")
print("Strip", "|"+string.strip()+"|")
```

Output:

```
Without strip: |  Toy Story     |
Left Strip |Toy Story   |
Right Strip |  Toy Story|
Strip |Toy Story|
```

translate():

This method uses a table to map the characters from source to target. The table can contain the ascii values and `None`.

```
# Create a translation table i -> I and a -> A
d = {105:73, 97:65}
string = "i saw a program. a bug was there."

# Map each matching character to the given value
print(string.translate(d))
```

Output:

```
I sAw A prograAm. A bug wAs there.
```

format():

This method is used to get rid of the concatenations. We can simply write the string with placeholders and pass the values that will be pasted to the specified places.

The placeholders are written using `{ }` and they can have a name, index or they can be empty too.

- The named placeholders are recognized by their names:

```
# Named Placeholder
string = "Hi {user}, welcome back to your {platform} desktop."
print(string.format(platform="Linux", user="Jack"))
```

Output:

```
Hi Jack, welcome back to your Linux desktop.
```

- The indexed placeholder are recognized by their indices.

```
# Indexed Placeholder
string = "Hi {1}, welcome back to your {0} desktop."
print(string.format("Linux","Jack"))
```

Output:

```
Hi Jack, welcome back to your Linux desktop.
```

> Note that first argument is mapped to `{0}` and second is mapped to `{1}.`

- The blank placeholder depends totally on position. If we change the position of arguments, they will be mapped accordingly.

```
# Blank Placeholder
string = "Hi {}, welcome back to your {} desktop."
print(string.format("Linux","Jack"))
```

Output:

```
Hi Linux, welcome back to your Jack desktop.
```

Checking String types:

The methods below can be used to check the type of strings:

startswith() & endswith():

We can check whether a string starts or ends with the given substring or not.

Code Block

```
s = "start-middle-end"
print("Starts with 'start':", s.startswith('start'))
print("Ends with 'end':", s.endswith('end'))
print("Starts with 'middle'", s.startswith('middle'))

```

Output:

Code Block

```
Starts with 'start': True
Ends with 'end': True
Starts with 'middle' False

```

isalpha(), isdigit() & isalnum():

`isalpha()` returns `True` if the string contains only alphabets.

`isdigit()` returns `True` if the string contain only numbers.

`isalnum()` returns `True` if the string is alphanumeric, that means it contains alphabets, digits or both.

Code Block

```
onlyAlphabets = "abcdefABCDEF"
onlyDigits = "012345"
alphanumeric = "abcd123"

print(onlyAlphabets.isalpha())
print(onlyDigits.isalpha())
print(onlyDigits.isdigit())
print(alphanumeric.isalnum())

```

Output:

Code Block

```
True
False
True
True

```

> **Note:** There are two additional methods `isdecimal()` and `isnumeric()`. These methods work same but their differences are rarely used in real world programs. The `isnumeric()` method is able to distinguish any string that is numeric, fractional etc. while `isdecimal()` can check for decimal numbers.

islower(), isupper() & istitle():

If the string contains only lowercase characters `a-z`, `islower()` will return `True`. Similarly `isupper()` will return `True` if the string contains only uppercase characters `A-Z`. If the string is formatted in a Camel Case fashion, `istitle()` will return `True`.

Code Block

```
lower = "cat and dogs"
upper = "DEER IN A FOREST"
camelcase = "Hello World"
print(lower.islower())
print(upper.isupper())
print(camelcase.istitle())

```

Output:

Code Block

```
True
True
True

```

Searching for substrings:

We can search for the substrings within a string using the following methods:

find() & rfind():

`find()` method returns the index of the first matching substring, when searched from left. If the string is not present, it returns `-1`.

The `rfind()` method works same but it searches for the substring from the right.

Code Block

```
s = "the quick brown fox jumped over the lazy dog"
print("'quick' present at:", s.find("quick"))
print("'the' present at:", s.find("the"))
print("'the' present at:", s.rfind("the"))
print("'cat' present at:", s.find("cat"))

```

Output:

Code Block

```
'quick' present at: 4
'the' present at: 0
'the' present at: 32
'cat' present at: -1

```

index() & rindex():

These method are similar to `find()` and `rfind()` methods but the only difference is that when the string is not found, `index()` and `rindex()` will raise an `ValueError`.

Code Block

```
s = "the quick brown fox jumped over the lazy dog"
print("'quick' present at:", s.index("quick"))
print("'the' present at:", s.index("the"))
print("'the' present at:", s.rindex("the"))
print("'cat' present at:", s.index("cat"))

```

Output:

Code Block

```
'quick' present at: 4
'the' present at: 0
'the' present at: 32
ValueError: substring not found

```

Splitting strings:

The below methods are used to split the string over the provided substring:

partition() & rpartition():

`partition()` method splits the string over the provided substring and returns a `tuple` of three values - the part before match, the matching substring and the part after match.

The `rpartition()` method is same as `partition()` but it searches for the substring starting from the right.

Code Block

```
s = "before match after"
print(s.partition('match'))

s = "Love For All, Hatred For None."
print(s.partition('For'))

s = "Love For All, Hatred For None."
print(s.rpartition('For'))

```

Output:

Code Block

```
('before ', 'match', ' after')
('Love ', 'For', ' All, Hatred For None.')
('Love For All, Hatred ', 'For', ' None.')

```

split() and rsplit():

The `split()` method returns a list of strings that is obtained after splitting the whole string from every matching substring.

Code Block

```
s = "a-b-c-d-e-f"
print(s.split('-'))

```

Output:

Code Block

```
['a', 'b', 'c', 'd', 'e', 'f']

```

The passed substring is optional and it is a blank space by default. We can also limit the number of splits by passing the second parameter which should be an `int`. By default, there is no limit on the number of splits.

Code Block

```
s = "Change the world by being yourself"
print(s.split())
print(s.split(' ', 3))

```

Output:

Code Block

```
['Change', 'the', 'world', 'by', 'being', 'yourself']
['Change', 'the', 'world', 'by being yourself']

```

> The `rsplit()` method is same when used with single or zero arguments. When we use the number of splits that is the second argument, the splits are done starting from right.

splitlines():

This method also returns a list of string but splits on the basis of new line characters `\n`. It is similar to `str.split('\n')`.

Code Block

```
s = '''Apple
Banana
Cherry
'''
print(s.splitlines())

```

Output:

Code Block

```
['Apple', 'Banana', 'Cherry']

```

The join() method:

We can join a list of strings using a separator string. Observe the example below:

Code Block

```
fruits = ['Apple', 'Cherry', 'Guava', 'Grapes']
print(", ".join(fruits))

```

Output:

Code Block

```
Apple, Cherry, Guava, Grapes

```

We can see that the list was joined using the `, ` string as the separator.

The replace() method:

This method replaces all the occurrences of source substring with the target substring.

Code Block

```
s = "Aspire to inspire before we expire"
print(s.replace('pire', 'PIRE'))

```

Output:

Code Block

```
AsPIRE to insPIRE before we exPIRE
```

The count() method:

This method returns the frequency of given character or substring in the string. We can optionally provide a start and end index to count in a particular substring.

Code Block

```
s = "mississippi"
print(s.count('s'))
print(s.count('ii'))
# Count from index 4 to 8
print(s.count('i', 4, 9))

```

Output:

Code Block

```
4
4
2

```

...

The encode() method:

This method converts the string into different encoding schemes. By default it uses `utf-8` but we can choose the encoding as per our need.

Code Block

```
text = "This is an example"
print("In utf-8:", text.encode())
print("In utf-8:", text.encode('utf-8'))
print("In utf-16:", text.encode('utf-16'))
print("In utf-32:", text.encode('utf-32'))
print("In ascii:", text.encode('ascii'))

```

Output:

Code Block

```
In utf-8: b'This is an example'
In utf-8: b'This is an example'
In utf-16: b'\xff\xfeT\x00h\x00i\x00s\x00 \x00i\x00s\x00 \x00a\x00n\x00 \x00e\x00x\x00a\x00m\x00p\x00l\x00e\x
In utf-32: b'\xff\xfe\x00\x00T\x00\x00\x00h\x00\x00\x00i\x00\x00\x00s\x00\x00\x00 \x00\x00\x00i\x00\x00\x00s\
In ascii: b'This is an example'

```

...

In this chapter, we learned the `str` class and its methods. `str` class is immutable, but we have plenty of methods available that we can use to write efficient programs with high level of string manipulation. In the upcoming chapter, we will learn about Dictionaries, another collection available in python via the `dict` class.

Chapter 13: Python Dictionary

Dictionary is one of the four collections in python. Other collections are List (Array), Set and Tuple. A dictionary contains key-value pairs inside curly braces `{}` where the key can be any immutable data structure. We can access the values using a unique key. For example:

Code Block

```
d = {'id':100, 'name':'joe', 100:'jack'}
# Here d is the dictionary with key value pairs

```

We can use dictionary using the `dict` class. The key-value pairs stored in the dictionary are unordered. But from `Python 3.7`, the dictionaries are able to remember the order of insertion. Dictionaries are mutable, that means we can easily manipulate it. In this chapter, we will explore the `dict` class and its methods to grasp a better understanding about the dictionaries.

Initialization of Dictionary:

We can initialize a dictionary using the below methods:

1. We can initialize the dictionary using the “key:value” pairs inside the curly braces `{ }`.

Code Block

```
d = {'Hello':'World', 100:101, 'Name':'Python', 'version':3.8}
print(d)
print(type(d))

```

Output:

Code Block

```
{'Hello': 'World', 100: 101, 'Name': 'Python', 'version': 3.8}
<class 'dict'>

```

1. We can also use the

`dict()` constructor to initialize a dictionary. We can pass a list of tuples into the constructor to create the dictionary.

Code Block

```
list_of_tuples = [('Hello', 'World'), (100,101), ('Name', 'Python'), ('version',3.8)]
d = dict(list_of_tuples)
print(d)
print(type(d))

```

Output:

Code Block

```
{'Hello': 'World', 100: 101, 'Name': 'Python', 'version': 3.8}
<class 'dict'>

```

1. We can also initialize an empty dictionary using empty curly braces

`{ }` or `dict()` constructor with no arguments.

Code Block

```
d = {} # Or d = dict()
print(d)
print(type(d))

```

Output:

Code Block

```
{}
<class 'dict'>

```

> Q: Can a key be of any data type?
>
> A: The answer is, “No”. A key can only be any immutable data structures like int, str, tuple etc.
>
> This is because the keys are hashed before mapping to a value. An immutable data structure will not be manipulated at all, hence the hash value will be same. However, if we use a mutable data structure, the hash value will change if someone manipulates the key and we won’t be able to find the corresponding value.

...

Accessing the values:

We can access the values using the keys inside a square bracket `[ ]` just like we do in case of arrays.

Code Block

```
d = {"Name":"Joe", "Age":30, "Id":1001}
print(d["Name"])
print(d["Id"])

```

Output:

Code Block

```
Joe
1001

```

> We can access any key directly. However we can not access a key that is not present. Python will raise a `KeyError` in such cases.

There is also a `get()` method available to fetch the value of a given key from the dictionary.

Code Block

```
d = {"Name":"Joe", "Age":30, "Id":1001}
print(d.get("Name"))
print(d.get("Id"))

```

Output:

Code Block

```
Joe
1001

```

Q: Can we have duplicate keys in dictionary?

A: There can be only single unique key in the dictionary. If we want to add a new value over an existing key, or we insert a duplicate key value pair, the previous value will be overwritten by this new value.

Assigning value to a key:

We can assign a value to a key just like we do in case of arrays. However, it is not necessary for the key to exist in dictionary beforehand in this case.

If the key already exists, its value will get updated. If it doesn't, the new key-value pair will be added.

Code Block

```
d = {"Id":100}
d["Country"] = "India"
d["Id"] = 101
print(d)

```

Output:

Code Block

```
{'Id': 101, 'Country': 'India'}

```

...

Size of Dictionary:

We can use the inbuilt `len()` function to get the number of key-value pairs present in the dictionary.

Code Block

```
d = {
    1:'Sun',
```

```
    2:'Moon',
    3:'Earth'
}
print("Size of dictionary is:", len(d))

```

Output:

Code Block

```
Size of dictionary is: 3

```

...

Membership in dictionary:

We can check whether a key exists in a dictionary or not using the `in` operator.

Code Block

```
d = {
    2:"Even",
    3:"Odd",
    4:"Even",
    16:"Even"
}
print("3 is in dictionary?", 3 in d)

```

Output:

Code Block

```
3 is in dictionary? True

```

...

Q: What is the complexity of searching for a key in the dictionary?

A: It depends on the types of keys present in the dictionary. The hashing algorithm used by dictionary can search in constant time in best case. However it can take linear time as well depending on the number of keys present. But that would be a rare case.

Looping over a dictionary:

A dictionary can be iterated using the `for...in` loop. By default, we can loop over the keys. However there are methods like `values()` and `items()` to loop over values or key-value pairs too. We will see them in the later sections.

Code Block

```
d = {'Math': 100, 'English': 90, 'Science': 95}
for x in d:
```

```
    print ("Key=", x, "Value=", d[x])

```

Output:

Code Block

```
Key= Math Value= 100
Key= English Value= 90
Key= Science Value= 95

```

...

Methods of Dictionary:

Dictionary supports some methods that makes it easier to manipulate it and use it for various real life problems. We will discuss all the popular and most used method in detail one by one.

Method Name	Description
update(d)	Updates the dictionary with the specified key-value pairs *d*.
setdefault(key, default_value)	Returns the value of passed key and updates the default value if the key does not exists
clear()	clears the dictionary
pop(key)	removes and returns value with the specified key
get()	Returns the value of specified key
popitem()	Returns and removes the last inserted pair
copy()	Returns a copy of the dictionary
fromkeys(keys, value)	Creates and returns a new dictionary with the specified keys and value.
items()	Returns an iterable with key-value tuple
keys()	Returns an iterable with keys only
values()	Returns an iterable with all the values

Lets see these methods with some examples.

1. update(d):

This method will update the dictionary using the passed dictionary (key - value pairs).

Code Block

```
d = {
    'Monday':1,
    'Tuesday':2,
    'Sunday':3
}
```

```
d.update({'Sunday':0, 'Wednesday':3})
print(d)
```

Output:

Code Block

```
{'Monday': 1, 'Tuesday': 2, 'Sunday': 0, 'Wednesday': 3}
```

2. setdefault(key, default_value):

This method is used to bypass the `KeyError` when we try to access a key that is not available. The setdefault() method returns the value of the specified key, if available. Otherwise it returns the default value provided as the second argument as well as updates it into the dictionary.

Code Block

```
d = {
    'Name':'George',
    'age':20
}
print(d.setdefault('age',25))
print("The dictionary after line 5",d)
print(d.setdefault('salary', 25000))
print("The dictionary after line 7",d)
```

Output:

Code Block

```
20
The dictionary after line 5 {'Name': 'George', 'age': 20}
25000
The dictionary after line 7 {'Name': 'George', 'age': 20, 'salary': 25000}
```

3. clear():

This method removes all the key-value pairs from the dictionary.

Code Block

```
d = {
    2:"Even",
    3:"Odd",
    4:"Even",
    16:"Even"
}
d.clear()
print(d)
```

Output:

Code Block

```
{}

```

4. pop(key):

This method removes the pair of the specified key and returns the value. A `KeyError` is raised if the key does not exist.

Code Block

```
d = {
    1:100,
    2:200,
    3:300,
    4:400
}
print(d.pop(1))
print("Dictionary after pop:", d)

```

Output:

Code Block

```
100
Dictionary after pop: {2: 200, 3: 300, 4: 400}

```

> We can also delete a key value pair using the `del` keyword.

Code Block

```
d = {1:'One', 2:'Two'}
del d[1]
print(d)

```

Output:

Code Block

```
{2:'Two'}
```

5. get(key):

This method returns the value of the specified key.

Code Block

```
d = {"Name":"Joe", "Age":30, "Id":1001}
print(d.get("Name"))
print(d.get("Id"))

```

Output:

Code Block

```
Joe
1001

```

6. popitem():

This method removes and returns the last inserted key-value pair in the dictionary. A `KeyError` is raised if the dictionary is empty.

Code Block

```
d = {
    1:100,
    2:200,
    3:300,
    4:400
}
print(d.popitem())
print(d)
```

Output:

Code Block

```
(4, 400)
{1:100, 2:200, 3:300}
```

7. copy():

This method returns the copy of the given dictionary.

Code Block

Ou

```
d = {2: "Even",
     3: "Odd",
     4: "Even",
     16: "Even"
     }
```

Code Block

```
{2: 'Even', 3: 'Odd', 4: 'Even', 16: 'Even'}

```

8. fromkeys(keys, value):

This method creates a dictionary using an iterable of keys. We can optionally pass a second argument which will be the default value of all the keys. If not provided, `None` is used.

Code Block

```
keys = ["Sun", "Mon", "Tue", "Wed"]
value = "day"
print(dict.fromkeys(keys, value))

```

Output:

Code Block

```
{'Sun': 'day', 'Mon': 'day', 'Tue': 'day', 'Wed': 'day'}
```

9. items():

This method returns an iterable of all the key value pairs in the dictionary. We can use it to get the list of tuples from the dictionary.

Code Block

```
d = {
    1: 'One',
    2: 'Two',
    3: 'Three',
    4: 'Four',
    5: 'Five'
}

for k, v in d.items():
    print("Key = %s value = %s"%(k,v))


```

Output:

Code Block

```
Key = 1 value = One
Key = 2 value = Two
Key = 3 value = Three
Key = 4 value = Four
```

```
Key = 5 value = Five

```

10. keys():

This method returns an iterable of all the keys in the dictionary.

Code Block

```
d = {
    1: 'One',
    2: 'Two',
    3: 'Three',
    4: 'Four',
    5: 'Five'
}

for k in d.keys():
    print(k)

```

Output:

Code Block

```
1
2
3
4
5

```

11. values():

This method returns an iterable of all the values in the dictionary.

Code Block

```
d = {
    1: 'One',
    2: 'Two',
    3: 'Three',
    4: 'Four',
    5: 'Five'
}

for v in d.values():
    print(v)

```

Output:

Code Block

```
One
Two
Three
Four
Five
```

Q: How do you check whether a value exists in a dictionary or not?

A: We can check it using the `values()` method like this:

`exists = True if value in d.values() else False`

Nested Dictionaries:

When we insert a dictionary inside another dictionary, it is called nested dictionary. Dictionary is mutable, hence it can't work as a key. But it can work as the value and further the value can have a key with another dictionary as the value. See the example below:

Code Block

```
d = {
    'name':{
        'first':'Joe',
        'last':'George'
    },
    'age':40,
    'address':{
        'line1':'43/1 Block',
        'line2':'New York',
        'contact':{
            'primary':'91-xxx-002',
            'secondary':'90-xxx-452'
        }
    }
}

```

...

Creating JSON from dictionary:

JSON (Javascript Object Notation) is a popular file format that is used for exchanging data and establishing communication between different languages. JSON also contains key value pairs just like dictionary. See an example of JSON:

Code Block

```
{
 "name":"Joe",
 "age":20,
 "address":{
   "street":"43",
   "city":"New York"
 }
}

```

Suppose we want to exchange data between two languages - Java and Python. Java has different collections than Python. In this situation they need a common notation - JSON to share the data. Python comes with a module to convert dictionary to JSON and vice versa. See the example below:

Code Block

```
# Import the json module
import json

# Create a dictionary
d = {
    'name':{
        'first':'Joe',
        'last':'George'
    },
    'age':40,
    'address':{
        'line1':'43/1 Block',
        'line2':'New York',
        'contact':{
            'primary':'91-xxx-002',
            'secondary':'90-xxx-452'
        }
    }
}

# Convert the dictionary into JSON
JSON_string = json.dumps(d)

# Print the string
print(JSON_string)

```

Output:

Code Block

```
{"name": {"first": "Joe", "last": "George"}, "age": 40, "address": {"line1": "43/1 Block", "line2": "New York"
```

In the above example,

- The `import` keyword is used to import other libraries. Here `json` is the library we want to use.
- The `dumps()` function converts the provided object to JSON and returns it as a string.

We can convert it back to `dict` using other function of `json` module. See the code below:

Code Block

```
# Convert the string back to dictionary
JSON_string = '{"name": {"first": "Joe", "last": "George"}, "age": 40, "address": {"line1": "43/1 Block", "li

dictionary = json.loads(JSON_string)

# Print the dictionary
print(dictionary)

```

Output:

Code Block

```
{'name': {'first': 'Joe', 'last': 'George'}, 'age': 40, 'address': {'line1': '43/1 Block', 'line2': 'New York
```

The `loads()` function is the opposite of `dumps()` function. It takes the JSON string as the argument and returns the dictionary back.

Now we can see that how a `dict` is different from other collections in python.

- A `list` contains ordered elements and allows duplicates. It is mutable.
- A `set` contains unordered elements and doesn't allow duplicates. It is immutable.
- A `dict` contains ordered elements from python 3.7 and its keys are non-duplicate. It is mutable.
- A `tuple` contains ordered items and allows duplicates. But it is immutable.

In this chapter, we learned how to create and manipulate dictionaries. We also learned how to access and modify the key value pairs as well as the methods of the `dict` class. In the next chapter, we will learn about the `tuple` and everything related to it.

Chapter 14: Python Tuple

Tuple is another collection in python which is ordered and immutable(can not be altered). It allows duplicate elements. It is one of the 4 collections available in python. Others are List, Dictionary and Set which are used in different scenarios. We can use the `tuple` class to create and use a tuple collection. Tuples are just like lists, but the elements inside them can not be reassigned or removed. In this chapter, we will explore the `tuple` class and its methods in details.

Initialization of a tuple:

We can initialize a tuple in following ways:

1. A tuple can be written as comma separated elements inside round brackets `( )`.

```
t = (1,2,3,4)
print(t)
print(type(t))
```

Output:

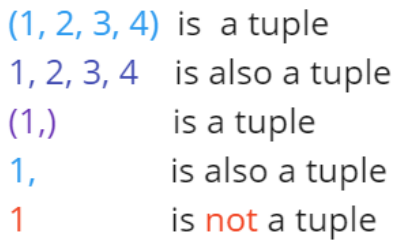
```
(1, 2, 3, 4)
<class 'tuple'>
```

However, the round brackets are not mandatory. The tuples are recognised through the `,` operator. Hence, we can omit the brackets `( )`. But including the brackets is considered as the best practice.

```
t = 1,2,3,4
print(t)
print(type(t))
```

Output:

```
(1, 2, 3, 4)
<class 'tuple'>
```



1. We can pass an iterable in the constructor of

`tuple` class. It will return a new instance of tuple with the passed values.

```
t = tuple(['cow','goat','rabbit'])
print(t)

t = tuple("PYTHON")
print(t)
```

Output:

```
('cow', 'goat', 'rabbit')
('P', 'Y', 'T', 'H', 'O', 'N')
```

1. We can initialize an empty tuple by using the

`tuple()` constructor with no parameters or empty round brackets `( )`.

```
t = ()  # OR t = tuple()
print(t)
print(type(t))
```

Output:

```
()
<class 'tuple'>
```

1. To initialize a tuple with single element, we can either use the

`tuple()` constructor or we can declare it with a following `,` operator.

```
t = 2,
s = (3,)
print(t)
print(s)
```

Output:

```
(2,)
(3,)
```

> Note: We can not declare a single element tuple like `t = (2)` as it will represent the value `2` only. Hence the `,` operator is mandatory.

No. of elements	Initializing with Constructor	Initializing with Literal
0	`t = tuple()`	`t = ()`
1	`t = tuple([1])`	`t = 1,` `t = (1,)`
More than 1	`t = tuple([1,2,3...])`	`t = (1,2,3…)` `t = 1,2,3…`

The above table shows various ways to create a tuple for different numbers of elements.

Accessing data from tuple:

A tuple is similar to `list`, so we can access the elements using their indices. We have to write the index inside the square brackets `[ ]`.

```
t = ('chain', 'lock', 'key', 'lid')
print(t[0])
print(t[2])
```

Output:

```
chain
key
```

Tuple also supports negative indices.

```
t = ('chain', 'lock', 'key', 'lid')
print(t[-1])
print(t[-2])
```

Output:

```
lid
key
```

Just like any iterable, tuple also supports ranged indexing.

```
t = ('chain', 'lock', 'key', 'lid')
print(t[0:3])
print(t[1:3])
print(t[0:3:2])
```

Output:

```
('chain', 'lock', 'key')
('lock', 'key')
('chain', 'key')
```

Looping over a tuple:

1. We can loop over a tuple using simple

`for` loop.

Code Block

```
birds = 'parrot', 'crow', 'eagle'
for bird in birds:
    print(bird, end=" ")

```

Output:

Code Block

```
parrot crow eagle

```

1. We can use while loop to iterate over the indices.

Code Block

```
birds = 'parrot', 'crow', 'eagle'
i = 0
```

```
while i<len(birds):
    bird = birds[i]
    print(bird, end=" ")
    i+=1

```

Output:

Code Block

```
parrot crow eagle

```

...

Membership in tuple:

We can determine whether an element/object exists in the tuple or not by using `in` operator.

Code Block

```
brands = 'honda', 'suzuki', 'BMW', 'Tata'

print("Maruti Present?", 'Maruti' in brands)
print("suzuki present?", 'suzuki' in brands)

```

Output:

Code Block

```
Maruti Present? False
suzuki present? True

```

> Note: The `in` operator searches for the elements linearly. Hence it takes linear time to check the membership.

...

Modifying a tuple:

Tuples are immutable, so we can not directly modify them. If we try to do so, we will get an `TypeError`.

Code Block

```
t = ('Summer', 'Winter', 'Spring')
t[0] = 'Fall'

```

Output:

Code Block

```
TypeError: 'tuple' object does not support item assignment
```

```
2
```

However we can modify the object that is present there.

Code Block

```
t = ([1,2,3], [4,5,6], [7,8])
# t[2] = [7,8,9] will raise an error
# But we can use list method to modify it
t[2].append(9)
print("Modified Tuple:", t)

```

Output:

Code Block

```
Modified Tuple: ([1, 2, 3], [4, 5, 6], [7, 8, 9])
```

This raises a question that how are the tuples immutable if we can change the content indirectly? New programmers often get confused when it comes to the immutability of the tuples. Lets understand how it works:

1. In the variables chapter, we saw that variable is merely an pointer that points to a physical object. Just like that, the elements of the tuple are references to the actual objects.

...

1. We can not change the position, value of these elements. However, we can modify the referred object, because it does not reside inside the tuple.

A clean way to modify a tuple is, first convert it into a list, modify it and then convert it back to tuple.

Code Block

```
t = ('Summer', 'Winter', 'Spring')
t = list(t)
t[0] = 'Fall'
t = tuple(t)
print(t)

```

Output:

Code Block

```
('Fall', 'Winter', 'Spring')

```

...

Size of tuple:

The `len()` function determines the number of elements present in the tuple.

Code Block

```
t = 1, 2, 3, 4, 5, 6
print("Length is :", len(t))

```

Output:

Code Block

```
Length is : 6

```

…

Unpacking a tuple:

Observe the below program:

Code Block

```
a = 1
b = 2
c = 3
print(a+b+c)

```

The above program can be written in a concise manner with the help of tuple.

Code Block

```
a, b, c = 1, 2, 3
print(a+b+c)

```

Here, in line 1, the right hand side is a tuple. In the left hand side, there are variables that are capturing the unpacked value. This is called tuple unpacking.

If the tuple has lesser or more values than the number of variables, a `ValueError` will be raised. However, we can overcome this by writing a asterisk operator with a variable in the last.

Code Block

```
t = (100,101,102,103)

a, b, *c = t
print(a, b, c)

a, b, c, d, *e = t
print(a, b, c, d, e)

```

Output:

Code Block

```
100 101 [102, 103]
100 101 102 103 []

```

We can see that the unmatched values are automatically assigned to the variable with `*` operator. We have seen this operator in the Functions chapter. There also such variable represents an iterable.

...

Methods of tuple:

There are only 2 methods available in the `tuple` class.

Method Name	Description
count(value)	Returns the count of specified value.
index	Returns the index of specified value.

1. count(value):

This method counts exactly how many times a value is present in the tuple.

Code Block

```
t = (1,1,2,3,2,1,3,2)
print(t.count(1))

```

Output:

Code Block

```
3

```

2. index(value):

This method returns the first index from left where the specified value is present. We can optionally provide a start and end point as second and third parameter to restrict the search range. If the provided value is not present, a `ValueError` will be raised.

Code Block

```
t = (100,101,102,103,100, 101)

print(t.index(100))

# Search in range from 2 to 6
print(t.index(101, 2, 6))

```

Output:

Code Block

```
0
5

```

...

Operators of tuple:

A tuple supports two types of operators (that `str` and `list` also support).

1. The `+` operator:

This operator is used between two or more tuples. This operator simply appends all the tuples together and returns a new tuple.

Code Block

```
tuple1 = 'Cow', 'rabbit', 'Goat'
tuple2 = 'Horse', 'Dog'
print(tuple1 + tuple2)

```

Output:

Code Block

```
('Cow', 'rabbit', 'Goat', 'Horse', 'Dog')

```

> Note that a tuple is immutable. Hence the original tuple will not be changed and a new copy will be returned.

2. The `*` operator:

The `*` operator is used between an `int` and a `tuple`. This operator, when used like `t*N` (where `t` is a tuple and `N` is an integer), will create exactly `N` copies of `t` and will append them together.

See the example below:

Code Block

```
t = (1,2,3)
N = 5
print(t * N)

```

Output:

Code Block

```
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)

```

...

Why tuple when we have list?

So far we have seen that there are so many restrictions on a tuple and it has lots of similarity to list. But why do we need tuple when we have the `list` class.

The answer lies in the applications of both. Lets see some details of both:

- A tuple is static in nature. That means it is defined once and uses limited amount of memory. On the other hand a list is dynamic and it often changes its size and memory.
- When we are working with a fixed sized collection, where modification is not required but only read operation is performed, tuple works efficiently. List will be slower in this case.
- But when we need all kinds of read modify operations, list is a better choice.
- Due to the immutable nature, there are less chances for any error to occur in case of tuples. While in list, we can often see such errors.
- List has more built in methods than tuple. But tuple can be converted to list to use those operations.
- Tuple consumes less memory and works faster. List consumes more memory due to its dynamic nature.

...

Now we can see that how a `tuple` is different from other collections in python.

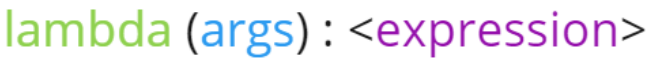
- A `list` contains ordered elements and allows duplicates. It is mutable.
- A `set` contains unordered elements and doesn't allow duplicates. It is immutable.
- A `dict` contains ordered elements from python 3.7 and its keys are non-duplicate.
- A `tuple` contains ordered items and allows duplicates. But it is immutable.

In this chapter, we learned how to create and use a tuple. We also saw how tuple is immutable and how the elements can be modified indirectly. We also saw some operations in tuple and its method. We learned why tuple is important despite having the `list` class. In the upcoming chapter, we will explore the lambda functions and will see some applications of lambda expressions in python.

Chapter 15: Lambda Functions

A Lambda function, also known as Anonymous Function, are functions with single liner body and no name. Such functions are declared using the `lambda` keyword. In this chapter, lets explore lambda functions in details.

Syntax of Lambda Expression:



Lets start by writing a simple function that returns the summation of two numbers:

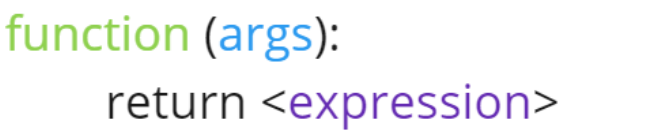
```
f = lambda (a,b): a + b
print("Sum of 2 and 7", f(2,5))
```

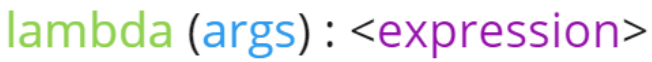
Output:

```
7
```

We can see that we have normally called the function using the round brackets `( )`.

Converting a normal function into lambda expression:





Note that:

1. A lambda function can have a single expression in its body.
2. The expression is evaluated and it is also the return value of the function.
3. There can be any number of parameters including 0.
4. The `return` keyword is not required.

Why to use a Lambda Function:

We already have our normal functions that are declared with the `def` keyword. The question arises, why do we need another kind of function? To understand it, lets consider a scenario.

We have already used the `sort` method of `list` class. To sort a list using our own comparator, we need to pass a reference to the comparator function. See the code below:

```
# Sort the following list on the basis of
# reversed strings.
l = ['ab','abb','baba','aabb','aba']

def compare_reversed(a):
    return a[::-1]

l.sort(key=compare_reversed)
print(l)
```

Output:

```
['aba', 'baba', 'ab', 'abb', 'aabb']
```

It was easy, because we just declared a function. But consider a scenario where we need to sort the lists in hundreds of methods. It is not a good practice to write separate functions for all the cases as we need to declare too many names.

Hence, a good practice is to use anonymous functions in such cases. See the code below:

```
# Sort the following list on the basis of
# reversed strings.
l = ['ab','abb','baba','aabb','aba']
l.sort(key=lambda x: x[::-1])
print(l)
```

Output:

```
['aba', 'baba', 'ab', 'abb', 'aabb']
```

We can see that the code became shorter and cleaner than before.

Map, Filter and Reduce:

There are some built in functions that are used frequently with lambda functions. These are `map()`, `reduce()` and `filter()` Lets see them one by one:

1. map(mapper_function, iterable):

The `map()` function takes two parameter: A mapper function and an iterable. The elements of the iterable are passed into the mapper – function and the result is stored sequentially in the form of another iterable. The mapper function must take a single value and must return a single value. Lets understand by an example:

Code Block

```
# Given a list of numbers, increase each number by 1
def increase_by_1(a):
    return a+1
```

```
l = [1,2,3,4]
r = map(increase_by_1, l) # returns a map object
print(list(r))
```

Output:

Code Block

```
[2, 3, 4, 5]
```

Again, we don't need to write a separate function every time we use map. See another example with lambda:

Code Block

```
# Given a list of strings, convert each string to upper case.
animals = ['dog','cat','rabbit']
upper_case_animals = map(lambda x: x.upper(), animals)
upper_case_animals = list(upper_case_animals)  # As map returns a map object
print(upper_case_animals)
```

Output:

Code Block

```
['DOG', 'CAT', 'RABBIT']
```

In the above code,

1. `lambda x: x.upper()` takes `x` and returns the uppercase version of it.
2. The map function applies this function on all the elements and creates a new list.
3. `dog` is mapped to `DOG`, `cat` is mapped to `CAT` and so on.
4. Finally we get the desired output in the form of a `map` object which can be converted into the `list` object.

2. filter(filter_function, iterable):

The `filter()` function takes a function and iterable as the arguments. The filter function must take a single value as input and must return a boolean value, on the basis of which the elements of the iterable are either kept or rejected. See an example:

Code Block

```
# Filter out the odd numbers from the given list
l = [4,21,5,61,3,23,2,6]
odds = filter(lambda x: x%2==1, l)
print(list(odds))
```

Output:

Code Block

```
[21, 5, 61, 3, 23]
```

In the above code,

1. `lambda x: x%2==1` is a boolean function which returns `True` if `x` is odd.
2. The `filter()` function will iterate through all the elements and check them using the lambda function.
3. The elements for which the function returns `True`, are added to the returned iterable.
4. The function returns a `filter` object that can be converted into `list` object.

3. reduce(reduce_function, iterable):

The `reduce()` function takes a reduce function and an iterable. The reduce function is a function that takes two arguments and returns a single result. This function takes the first two elements in the sequence and replaces them with the result of reduce function. Further, it takes the next element and the result we got in previous step and applies the function again. It goes on for the whole iterable and we get an accumulated result in the end.

The `reduce()` function is not available as a built-in function from python 3. However we can use the `functools` library.

See the example below:

Code Block

```
from functools import reduce

# Apply concatenation on below strings

l = ['F','r','u','i','t','s']
result = reduce(lambda x,y:x+y, l)
print("Combined Sequence is:", result)

```

Output:

Code Block

```
Combined Sequence is: Fruits

```

> In the statement `from functools import reduce` , `from` and `import` are keywords to import the external libraries. `functools` is a package and `reduce` is the function inside the package. We will get into details in a separate section.

In the above code,

1.

`lambda x,y:x+y` is a function that takes `x and y` and returns their concatenation `x+y`.

1. The list is sequence of characters on which the reduce function is applied.

1. The reduce function applies the lambda function in sequence, it takes the

`F` and `r` and returns `Fr` in the first step.

1. Further, it takes

`Fr` and `u` and returns `Fru`. It goes on and finally the function returns `Fruits`.

1. If the list is having a single element, it is returned. However it the iterable has no elements in it, a

`TypeError` is raised.

...

Using lambda functions with callbacks:

One of the most important usage of lambda is when we use callback functions. Consider the functions below:

Code Block

```
1  def functionA(N, callback):
2      l = []
3      for i in range(N):
4          l.append(i+1)
5      callback(l)
6
7  def functionB(l):
8      for i in l:
9          print(i)
10
11 def functionC(l, start, end):
12     for i in range(start, end):
13         print(l[i])
14
```

The `functionA` takes 2 arguments `N` and `callback`. Here, `callback` is a function, which should take a single argument as we can see in line 5. The functions that are passed inside other functions and called at the end are called callback functions.

We can use `functionB` as the callback function as it takes the exact same number of arguments:

Code Block

```
1  functionA(5, functionB)
2
```

Output:

Code Block

```
1  1
2  2
3  3
4  4
5  5
6
```

But the problem is, what if we want to call `functionC` instead? We can not pass the extra arguments directly. One solution is to change the code of `functionA` and pass the extra arguments. This scenario will come in projects most of the time because the libraries we will use are not always designed as per our need and we cannot change the source code.

The solution is to use the lambda functions. We can pass a lambda function designed in such a way that it calls the required function with our arguments:

Code Block

```
start, end = 3, 7
functionA(10, lambda l:functionC(l, start, end))

```

Output:

Code Block

```
4
5
6
7

```

When the callback is called, it passes the argument `l` to the lambda function, that again uses it with extra arguments to call `functionC`.

...

In this chapter, we have learned how to use lambda function as well as some real life scenarios where lambda function works as a powerful tool. We also learned some of the built in functions map, filter and reduce. Remember that reduce is not available directly in python3. In the next chapter, we will explore class and objects in python.

Chapter 16: Class and Objects in Python

Class:

A class is a blueprint of a real world entity. Just like a machine, a house or any entity, we need a blueprint or map, we need a class to create an object. In Python, we use the `class` keyword to create a class. A class can have properties and methods. See the syntax below:

...

Lets see an example:

Code Block

```
class Book:
    title = "The Ickabog"
    author = "J K Rowling"

    def print_book(self):
        print("Title = {} , Author = {}".format(self.title, self.author))

```

In the above class,

- `Book` is the name of the class and `title` and `author` are the properties of this class.
- The `print_book` looks like a function but it is called a “method”.
- The first parameter of the method is `self` which represents the object itself. We will talk about `self` in a little bit later.
- The body of this method is the action that the object of this class will perform.

In python, there is no restriction on file name in which a class is saved. It can have any name. To use a class, we need objects. Lets see how to create and use objects.

...

Object:

An object is simply an instance of the class. Just like the blueprint exists on the paper, but the machine physically exists, the class exists in code but the object physically exists. Each Object has its own **State**, **Behavior** and **Identity**.

...

The ‘*state*’ of the object is represented by its properties. The ‘*behavior*’ is represented by its methods and the ‘*identity*’ is represented by its physical location in memory.'

We have been creating the objects in the recent chapters. To create an object, we need to use the class name with the parameters inside the round brackets `()`. See the example below:

Code Block

```
class Book:
    title = "The Ickabog"
    author = "J K Rowling"

    def print_book(self):
        print("Title = {} , Author = {}".format(self.title, self.author))

```

```
book = Book()
book.print_book()

```

Output:

Code Block

```
Title = The Ickabog , Author = J K Rowling

```

Here,

- The `Book()` statement creates a new Object into memory.
- It is assigned to a variable `book`. We can call the methods of this `book` object by using a `.` operator.
- This class has a single method. `book.print_book()` calls the method.
- We can also access the properties using `.` operator like `book.title`.

...

The Constructor:

A constructor is a method that is called upon the creation of the object. If we don't provide a constructor, python uses its default constructor. We can define a constructor using a magic method `__init__`. Lets understand with the help of an example:

Code Block

```
class A:
    def __init__(self):
        print("A new object created")

a = A()

```

Output:

Code Block

```
A new object created

```

We can see that as soon as the object was created, the `__init__()` method was called automatically. We can also pass parameters in the constructor.

Code Block

```
class B:
    def __init__(self, a, b):
        print("A new object created with initial args", a, b)

b = B(10, 20)

```

Output:

Code Block

```
A new object created with initial args 10 20

```

This way we can pass initial parameters upon the creation of the object.

...

Self:

The term `self` is not a keyword. In python, the first parameter of any instance method represents the instance of current class. It is used to refer the class variables and methods. We usually name it as `self` but we can use any other name instead of `self` if we want:

Code Block

```
class Printer:
    def show(a):
        print(a)

p = Printer()
print(p)
p.show()

```

Output:

Code Block

```
<__main__.Printer object at 0x000001266992DE20>
<__main__.Printer object at 0x000001266992DE20>

```

We can see that both `p` and the first parameter `a` represent same object. This is how `self` works.

We don't need to pass value of `self` to the method, python will do it automatically for us. So whenever we define a method, remember that `self` must be the first parameter.

`self` can also be used to access other methods and properties inside an class.

Code Block

```
class Book:
    title = None
    author = None

    def __init__(self, author, title):
        self.author = author
        self.title = title
        self.show()
    def show(self):
        print("Title = {} , Author = {}".format(self.title, self.author))


book = Book("Nora Roberts", "Legacy")

```

Output:

Code Block

```
Title = Legacy , Author = Nora Roberts

```

- We can see that initially the properties are `None`.
- In the constructor, we are setting the values of these properties through `self` in line 6 and 7.
- In line 8, we are calling another method in constructor through `self`.
- Note that in a class the order of the methods doesn't matter. So we can call other methods that are written after that statement.
- In line 10, we can see that we are able to access the properties in `print` statement.

...

Type of Methods:

There are basically 3 types of methods: instance methods, class methods and static methods. The ones we have seen so far are called instance methods. Lets see how they are different from each other.

1. Instance Method:

An instance method is bound to the object. Every object maintains its own state. Hence the behavior must also be different. The instance methods are specific to the objects and they can be called only by an instance of a class.

Code Block

```
class Vehicle:
    def __init__(self, name, state):
        self.name = name
        self.state = state

    def show_state(self):
        print(self.name, "is", self.state)

v1 = Vehicle("v1","Running")
v2 = Vehicle("v2","Stopped")
v1.show_state()
v2.show_state()

```

Output:

Code Block

```
v1 is Running
v2 is Stopped

```

We can see that each object has a different state. Hence the behavior is also different.

2. Static Method:

A static method does not depend on objects. It can be called through class name without creating an object. These methods do not depend on the state of an object, hence called static methods. They are more like functions, but called through the class name.

In python, the static method uses a decorator `@staticmethod` and does not need the `self` parameter. A decorator is a special annotation that enhances a function or method without any internal code changes. We will learn more about decorators in upcoming chapter. Since static methods do not have a `self` argument, it is not possible to read or modify the state of an object through static methods.

Code Block

```
class Pen:
    @staticmethod
    def show_available_colors():
        print("Blue", "Black", "Red")

Pen.show_available_colors()

```

Output:

Code Block

```
Blue Black Red

```

If we don't use the decorator above method definition, python will think of it as an instance method, and the first parameter will automatically become `self`.

3. Class methods:

A class method, just like static method, does not depend on the object of the class. We can call it just by using class name. However, the first argument refers to the class of the object. As an industry standard, we write it as `cls` but any other name will also work the same. We use `@classmethod` decorator to declare a method as class method.

Just like instance method can modify the state of the object, class method can modify the state of a class, through class variables, that will be reflected across all of the instances.

Code Block

```
class Bank:
    name = "National Bank"
    year = 2020

    @classmethod
    def change_year(cls):
        cls.year+=1

b1 = Bank()
b2 = Bank()
print(b1.year, b2.year)

b1.change_year()
print(b1.year, b2.year)

```

Output:

Code Block

```
2020 2020
2021 2021

```

In the above example, we can see that when we called the `change_year()` method, the `year` property changed for both of the objects.

> Q: What is the usage of class method?
>
> A: Class methods are used to share the values globally. If there is a global change, one does not need to modify all the objects but they can be changes via a class method call. Note that even if we call the method through any object, the changes will be global.
>
> Second, class methods are used to provide multiple constructors as per need. See the example below:

Code Block

```
class Person:
    name = None
    age = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_year_of_birth(cls, name, year):
        age = 2021 - year
        return cls(name, age)

    def print_details(self):
        print("Hi I am {}, {} years old.".format(self.name, self.age))

p1 = Person("Joe", 23)
p2 = Person.from_year_of_birth("Jack", 1994)
p1.print_details()
p2.print_details()

```

Output:

Code Block

```
Hi I am Joe, 23 years old.
Hi I am Jack, 27 years old.

```

> We can see that our program can have more than one type of constructor to create the objects according to the situation.

Creating / Deleting Object Properties:

In python, we can create or delete object properties without changing the code of the class. Just like we can access the properties through `.` operator, we can assign values to them and we can also create new ones for a particular object. Let's see how:

Code Block

```
class Seasons:
    spring = 0
    summer = 1
    fall = 2
    winter = 3

s = Seasons()
s.monsoon = 4
print(s.monsoon)

```

Output:

Code Block

```
4

```

We can see that even if `monsoon` property was not present before, we can assign a value to it. But note that it is not globally applicable and hence any other object won't have such property.

To delete a property, we can simply use the `del` keyword.

Code Block

```
class Seasons:
    spring = 0
    summer = 1
    fall = 2
    winter = 3

s = Seasons()
s.monsoon = 4
print(s.monsoon)
del s.monsoon
print(s.monsoon)

```

Output:

Code Block

```
4
AttributeError: 'Seasons' object has no attribute 'monsoon'

```

...

Creating empty class or methods:

Sometimes, we need to create an empty class or method without any body. These are needed when creating prototypes in team development. In python, we can use the `pass` keyword to indicate the empty body without getting a syntax error. See the example below:

Code Block

```
# Insert code to complete the below class
class Animal:
    pass

class Person:
    # Complete the code below
    def __init__(self, name, age):
        pass

```

> Note that we can create objects of empty classes without getting any error. They will be stateless but will have physical memory location. It is because in python, it is possible to assign properties through objects. Hence an empty class can also be used in a program.

In this chapter, we have seen how classes and objects are created. We also saw some python specific features of class like `self`, class method and creation of properties through objects. In the upcoming chapter, we will explore some object oriented concepts and how to implement them in python.

Chapter 17: Object Oriented Programming in Python

There are 3 fundamental concepts in object oriented programming - Inheritance, Encapsulation and Polymorphism. Inheritance enables us to write reusable classes, encapsulation secures our important data and polymorphism allows us to use different methods with same name. Lets explore these concepts in details in Python.

Inheritance:

Inheritance is a mechanism by which a class is able to use the properties and methods of another class without re-writing the implementation. Just like a child can inherit the properties of a parent - like estate, genital features and habits, a child class can inherit the behavior and state of a parent class.

In python, we can create a child class by mentioning the parent class name in `()` with its name. See the example below:

```
class Parent:
    def say(self):
        print("Hello I am the parent class")

class Child(Parent):
    pass

c = Child()
c.say()
```

Output:

```
Hello I am the parent class
```

In the above example,

- Class `Parent` is the parent or base class.
- `Child` class is the child or derived class and it inherits the `Parent` class because `Parent` class's name is written along with it.
- The object of `Child` class can use the method defined in the `Parent` class.

Suppose there is a class with hundreds of methods. If we want to add one more method without changing its code, we can simply inherit it and then write the additional method. This way we will be able to use all the old methods along with the new one.

```
class Artist:
    def sing(self):
        print("Sing")

    def act(self):
        print("Act")

class ChildArtist(Artist):
    def dance(self):
        print("Dance")

ca = ChildArtist()
ca.sing()
ca.act()
ca.dance()
```

Output:

```
Sing
Act
Dance
```

If we want to change the definition of any existing method, we can also do that with the help of inheritance.

```
class Rectangle:
    def has_four_sides(self):
        print("This shape has 4 sides")

    def area(self):
        print("Area is calculated using A*B ")

class Square(Rectangle):
    def area(self):
        print("Area is calculated using A*A ")

square = Square()
square.area()
square.has_four_sides()
```

Output:

```
Area is calculated using A*A
This shape has 4 sides
```

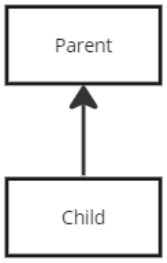
Redefining the same method name in child class is called dynamic polymorphism. We will see it in details in a bit.

Types of Inheritance:

Python supports four types of inheritance - Single, Multiple, Multilevel and Hierarchical. Lets see how to implement them with the help of the example:

1. Single Inheritance:

It is the simplest type of inheritance in which a class inherits another class.

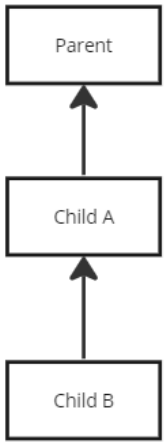



```
class Base:
    def method(self):
        print("This is base")

```

```
class Derived(Base):
    pass
```

2. Multileveled Inheritance:

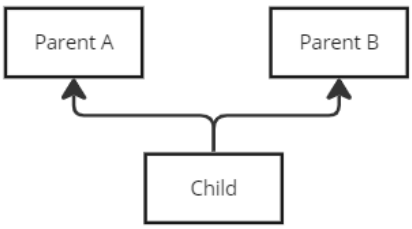
It is also known as chained inheritance. In multilevel inheritance, the parent class also has a parent and that class may further have another parent.




```
class A:
    pass

class B(A):
    pass

class C(B):
    pass
```

3. Multiple Inheritance:

A class can inherit more than one class. We call this multiple inheritance. All the classes being inherited are written using comma separator.



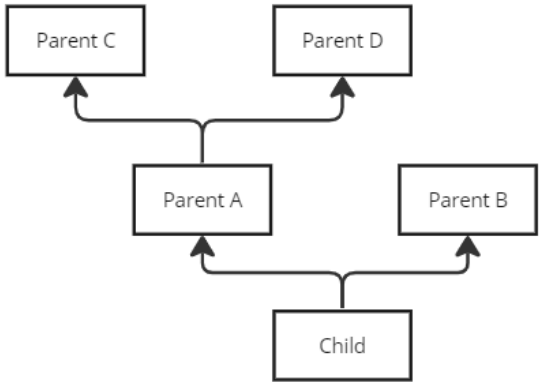

```
class A:
    pass

class B:
```

```
    pass

class C:
    pass

class D(A, B, C):
    pass
```

Method Resolution Order:

In multiple inheritance, there may be a situation that the parent classes are having same method name. In this situation, the problem is, how to know which method should be used. Due to this problem, languages like Java do not support multiple inheritance (although there is another method to achieve it.)

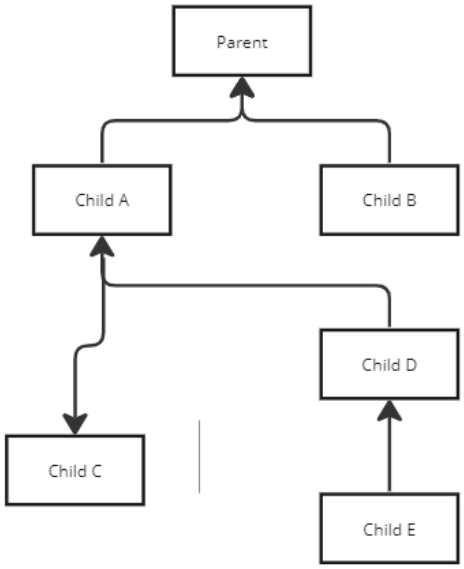
In python, the interpreter searches for the method in the BFS (Breadth First Search) order which is also known as Method Resolution Order or MRO. Suppose we have declared the class like `class A(B, C, D):` then it will search first in B, then in C and in D. If the method is still not found, the parents of B, C and D will be searched in the same manner. This is how python resolves the conflict and the method that hits first, is chosen.



Lookup Order: A, B, C, D

4. Hierarchical Inheritance:

It is the combination of all previously discussed inheritances:




```
class A:
    pass

class B(A):
    pass

class C:
    pass

class D(B, C):
    pass
```

Encapsulation:

Encapsulation means binding the state and behavior in a single entity. This entity is called Object. Encapsulation makes sure that our private data is hidden from outside world. When we use any third party library, the developer might not want us to change the logic by interfering to the members (methods or properties) of the class. In such scenario, we make the members private or protected. By default all the members are public and they can be accessed from anywhere.

Hiding data through private member:

In python, we can declare a private member using double underscores `__` in the property's or method's name. A private member can be accessed only within a class and no where else. Lets see an example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def show_details(self):
        print("My name is {} and I am {} years old.".format(self.name, self.__age))

person = Person("Jack", 20)
```

```
person.show_details()
print(person.__age)
```

Output:

```
My name is Jack and I am 20 years old.
AttributeError: 'Person' object has no attribute '__age'
```

We can see that the property `__age` is accessible within the class but it is not accessible outside the class.

> The magic methods like `__init__()` also start with double underscores and they are intended to be used inside a class. However they are not private and can be used outside as well.

Hiding Data through protected members:

The protected members should be accessed in the same class as well as the derived class. We place a single underscore `_` in front of the member's name. However python allows us to access protected members just like public members. Protected members are just used for coding convention and a programmer must not use them anywhere beside class or sub-class. Lets see an example:

```
class Animal:
    def __init__(self, name):
        self._name = name
    def _details(self):
        print("Animal name is", self._name)

class Dog(Animal):
    def details(self):
        self._details()

dog = Dog("Dog")
dog.details()
print(dog._name)
```

Output:

```
Animal name is Dog
Dog
```

See that we can easily access protected members from derived class as well as from outside of the class. But accessing protected members from outside is not a good coding practice and should be used for debugging purpose only.

> Note: Python also allows us to access private members using a special syntax. To access a private member `MemberName` of class `ClassName` through object `ObjectName`, we should write it like `ObjectName._ClassName__MemberName`. See the example below:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def show_details(self):
        print("My name is {} and I am {} years old.".format(self.name, self.__age))

person = Person("Jack", 20)
print(person._Person__age)
```

Output:

```
20
```

> We can see how we accessed the private member too. However, this method is not used in real world applications and it is limited to debugging purposes only. A private member is intended to be used inside a class only and a programmer must follow this. Otherwise it is considered a bad practice and may result in malfunctioning application.

Polymorphism:

Polymorphism is a mechanism through which we can use single entity to represent multiple functionalities. The literal meaning of polymorphism is to inhibit multiple forms through single object. In programming, polymorphism means using a single method name in multiple methods. There are multiple types of polymorphism - class polymorphism, method overriding etc.

In **class polymorphism**, we can use single method name in multiple classes. Lets see an example:

```
class Rabbit:
    def speed(self):
        print("Fast")

class Turtle:
    def speed(self):
        print("Slow")

rabbit = Rabbit()
turtle = Turtle()
rabbit.speed()
turtle.speed()
```

Output:

```
Fast
Slow
```

In **Method Overriding**, we can use the same name in child and parent class. When calling the method, it will be searched in the BFS order starting from the class of the object.

```
class A:
    def method(self):
        print("This is the method of class A")

class B(A):
    def method(self):
        print("This is the method of class B")

a = A()
b = B()
a.method()
b.method()
```

Output:

```
This is the method of class A
This is the method of class B
```

Python does not support Method Overloading. But we can achieve that in a different way.

```
class Shape:
    def area_circle(self, r):
        print(3.14*r*r)

    def area_square(self, a):
        print(a*a)

    def area_rectangle(self, a, b):
        print(a*b)

    def area(self, a, b=None):
        if b == None:
            if isinstance(a, float):
                return self.area_circle(a)
            elif isinstance(a, int):
                return self.area_square(a)
        else:
            return self.area_rectangle(a, b)

s = Shape()
# Area of circle
s.area(10.0)
# Area of Square
s.area(10)
# Area of Rectangle
s.area(10,20)
```

Output:

```
314.0
100
200
```

Here the inbuilt function `instanceof()` returns `True` if the first argument (object) is the instance/object of second argument (class). This way we can also achieve method overloading.

The super() function:

The `super()` function returns a reference to the parent class using which we can specifically invoke the parent class members. This function is useful when there is a method in child class with same name as parent class and we want to use the parent class method instead.

```
class A:
    def method(self):
        print("This is the method of class A")

class B(A):
    def method(self):
        super().method()
        print("This is the method of class B")


b = B()
b.method()
```

In this chapter, we learned about the fundamental concepts of Object Oriented Programming, We learned Inheritance and its types, multiple inheritance and the Method Resolution Order. We also learned Encapsulation and Polymorphism along with some python specific implementations. In the next chapter, we will talk about the scopes in python.

Chapter 18: Scope in Python

When we declare a variable, there is a limit to where the variable is visible. By visibility, we mean that the variable can be accessed or modified and this limit is called scope. A variable that is visible inside a block only, is called local variable and a variable that is visible every where is called global variable.

While coding a large application, it is important to know which variable is visible to which place. If we understand the concept, we will never struggle with bugs related to scope. In this chapter we will explore scope in python with some examples.

Local Variables and Local Scope:

The variables that are declared inside a function or method are called local variable. They are visible inside the function and the sub functions only. The body of the function is called local scope of this variable because the variable is visible only inside the function or method. See the example below:

```
var = "Python"

def foo():
    local_var = "Program"
    print("Inside foo: var=",var)
    print("Inside foo: local_var=",local_var)

foo()
print("Outside foo: var=",var)
print("Outside foo: local_var=",local_var)
```

Output:

```
Inside foo: var= Python
Inside foo: local_var= Program
Outside foo: var= Python
NameError: name 'local_var' is not defined
```

We can see that `local_var` is printed successfully inside the function but it raised a `NameError` when accessed from outside.

However, the functions are able to access the variables outside the body. We can see that `var` is visible both outside and inside a function.

Global Variables:

A variable declared in the main body is called global variable. A global variable can be read from a function. However we can not directly modify its value. See the example below:

```
global_variable = "Hello"

def foo():
    print(global_variable)
    global_variable = "World"
    print(global_variable)

print(global_variable)
foo()
print(global_variable)
```

Output:

```
Hello
UnboundLocalError: local variable 'global_variable' referenced before assignment
```

In the above code, line 4 raised the above error. The reason is, as soon as we tried to modify `global_variable` in line 5, python could not recognize it and declared a new variable with a new object.

So the `global_variable` you see inside the function is actually a new object with a local scope. Hence, in line 4, we can not read it before its declaration.

If we remove line 4, the program will work fine but function will not modify the `global_variable`.

```
global_variable = "Hello"

def foo():
    global_variable = "World"
    print("Inside foo",global_variable)

print("Before foo", global_variable)
foo()
print("After foo", global_variable)

```

Output:

```
Before foo Hello
Inside foo World
After foo Hello
```

The global keyword:

We can declare a global variable using the `global` keyword. It is useful when we want to use the same variable inside and outside the function. Lets see how:

```
global_variable = "Hello"

def foo():
    global global_variable
    print("Inside foo before modify", global_variable)
    global_variable = "World"
    print("Inside foo after modify",global_variable)

print("Before foo", global_variable)
foo()
print("After foo", global_variable)
```

Output:

```
Before foo Hello
Inside foo before modify Hello
Inside foo after modify World
After foo World
```

See that the variable declared global can be modified both outside and inside the function.

Enclosing scope:

When we declare a function inside function, they are called nested functions. The outer function is called enclosing function and the inner function is called enclosed function.

The enclosed function can access the variables declared in enclosing function. In simple words, the inner function can access the variables in outer function. However, it can not modify them.

```
def outer_f():
    x = 10
    def inner_f():
        print(x)
    print(x)
    inner_f()


outer_f()
```

Output:

```
10
10
```

Summary:

Finally we can observe few points:

1. A function can access variables from main body of a program. But it can not modify them.
2. An inner function can access variables from outer function and main body but it can not modify them.
3. If we try to change the value of a outer variable, a new local variable is created and treated accordingly.
4. To make a variable modifiable from both local and global scope, we use the `global` keyword.

Chapter 19: Modules in Python

When we are developing a program, the lines of code increase. The larger the program is, more lines of code are needed. After a time, the code becomes too large that it becomes difficult to handle. This is where we need modularization. In python, module is a way to manage our code by writing it into multiple files and using it where required. In general, we can say that using module we can create our own libraries that we can simply import in multiple places.

Creating a Module:

Modules are flat python files that contains classes and functions. But ideally they should not contain any code that is executed upon running that file. For example lets create a module that contains some mathematical functions. Create a file named “math_functions.py“ and write the following lines in it:

```
1  def add(x,y):
2      return x+y
3
4  def square(x):
5      return x*x
6
7
8  def factorial(x):
9      ans = 1
10     while x:
11         ans*=x
12         x-=1
13     return ans
```

If we try to run the above file, we won’t get any output because none of the function is being called. Now create another file with any name and write the following code in it:

```
1  import math_functions
2
3  print(math_functions.add(3,4))
4  print(math_functions.square(4))
5  print(math_functions.factorial(6))
```

Output:

```
1  7
2  16
3  720
```

In the above code:

1. `import` is a keyword that is used to use functions and classes present in other files.
2. We have imported the `math_functions` module because it was the name of the file that we gave.
3. All the functions are in their namespaces and they can be accessed through the module name. We can see how we used the functions in line 3, 4 and 5.

---

from … import statement:

We can import selected functions/classes/variables from the module using `from … import` statement.

- To import selected functions, we write their names only:

```
from math_functions import add

print(add(3,4))
```

Output:

```
7
```

We can directly use the function without using module name this way.

- We can import more than one function too. Note that we can use only specified functions only.

```
from math_functions import add, square

print("Add::", add(3,4))
print("Square::", square(6))
print("Factorial::", factorial(5))
```

Output:

```
Add:: 7
Square:: 36
NameError: name 'factorial' is not defined
```

We can see that using `factorial` raised an error because we did not import it.

- We can also import all the functions in single statement. We use the wildcard character `*` in such case.

```
from math_functions import *

print("Add::", add(3,4))
print("Square::", square(6))
print("Factorial::", factorial(5))
```

Output:

```
Add:: 7
Square:: 36
Factorial:: 120
```

Using an alias in module:

We can use `as` keyword to use a module with different name. It is useful when the modules are having large names. For example it is not very easy to type `math_functions` every time we want to use a function. See the example below:

```
import math_functions as mf

print("Add::", mf.add(3,4))
print("Square::", mf.square(6))
print("Factorial::", mf.factorial(5))
```

Output:

```
Add:: 7
Square:: 36
Factorial:: 120
```

The dir() function:

`dir()` is a built in function that gives the details of all the functions in the module.

```
import math_functions
print(dir(math_functions))
```

Output:

```
['builtins', 'cached', 'doc', 'file', 'loader', 'name', 'package', 'spec', 'add', 'factorial', 'square']
```

Built in modules:

Python is famous for its vast library. There are many function and classes available that can be used to write programs very easily. Some of the popular libraries are given below:

1. math
2. functools
3. collections
4. numpy
5. scipy
6. requests

There are many others and they are useful as per requirements. Lets see an example of how to use these modules:

1. math

```
import math

print(math.factorial(5))
print(math.sin(45))

```

Output:

```
120
0.8509035245341184
```

1. collections

```
from collections import Counter

c = Counter([1,2,2,3,2,1])
print(c)
```

Output:

```
Counter({2: 3, 1: 2, 3: 1})
```

Here in this example, we have imported `Counter` which is a data structure to store the frequency of elements in an iterable. It stores the frequency in form of dictionary where keys are the elements and values are the frequencies.

Testing a Module:

When we write a module, we have to test it so that it doesn't produce error while used in other places. Lets write a module and see how we can test its functionalities. Create a new file named `print_details.py` and add the following code into it:

Code Block

```
def printName(name):
    print("My name is {}.".format(name))

def printAge(age):
    print("I am {} years old.".format(age))

printName("Johan")
printAge(23)

```

When we run this file we get the following output:

Output:

Code Block

```
My name is Johan.
I am 23 years old.

```

Now we are sure that our module is working fine so we can import it into our other files. So lets write a new program and use the functions from our module:

Code Block

```
import print_details as pd

pd.printName("Robert")
pd.printAge(30)

```

Output:

Code Block

```
My name is Johan.
I am 23 years old.
My name is Robert.
I am 30 years old.

```

We can see that we got some unexpected output. It happened because the code we wrote for testing also executed. To prevent this from happening, we can do many things like removing test code or commenting it out but that is not a permanent solution.

There is a more efficient way, we can use the inbuilt `__name__` parameter. This parameter stores the name of the process and we can use it to prevent execution of our testing code when called from other python files.

When we execute a file directly, it is named as `"__main__"`. Lets modify our `print_details` module:

Code Block

```
def printName(name):
    print("My name is {}.".format(name))

def printAge(age):
    print("I am {} years old.".format(age))

if __name__ == '__main__':
    printName("Johan")
    printAge(23)

```

Output:

Code Block

```
My name is Johan.
I am 23 years old.

```

Now execute our main file from where we are calling this module:

Code Block

```
import print_details as pd
pd.printName("Robert")
pd.printAge(30)

```

Output:

Code Block

```
My name is Robert.
I am 30 years old.
```

Summary:

1. Module is a file to manage our code in large applications.
2. We can create our own modules and import them in other files.
3. Python has a large set of modules to make programming easy.

In this chapter, we saw how to create our own module and how to use them. We also explored importing functions from modules in different ways and using aliases.

Chapter 20: Iterators in Python

An iterator is an object that we can use to iterate over iterable objects or containers like list, tuple dictionaries etc. Iterator contains some values that we can iterate over using loops or `next()` function. In this chapter, lets explore iterators and see how to use them in our applications.

---

iter() and next():

Both `iter()` and `next()` are built-in functions. `iter()` is used with an iterable object like list, set, tuple etc. and it returns an iterator. We can use that iterator object with `next()` function to iterate over the values that the iterable was holding. Lets understand with an example.

```
list_of_number  = [2, 3, 5, 7, 11]

l_iterator = iter(list_of_number)

print(next(l_iterator))
print(next(l_iterator))
print(next(l_iterator))
print(next(l_iterator))
print(next(l_iterator))
```

Output:

```
2
3
5
7
11
```

In the above code:

1. `l` is the iterable that holds some values.
2. `l_iterator` is the iterator that we got by using `iter()` function. This object can be used with `next()` function to get the values one by one.
3. `next(l_iterator)` returns the values one by one.

> **Note:** We can also loop through the iterable using `for` loop. But `for` loops internally use the concept of iterators. They internally create an iterator and call the `next()` function for that. But to make things easier for programmers, such details are abstracted.

Iterator and Iterable:

Sometimes we may get confused between iterator and iterable. Lets observe few points to understand their differences:

- An iterator is an object that we get using `iter()` function over an iterable.
- Ideally an iterable is a container that we use to store the values but an iterator is a pointer to the values that moves upon calling the `next()` function.
- We can not directly use `next()` function on an iterable.
- For example a list, set etc. are iterables and they return an iterator when `iter()` used on them.

```
l = [1,2,3]         # Iterable
```

```
iterator = iter(l)  # Iterator
```

Creating an iterable:

We can create our own iterable. An iterable class contains two special methods: `__iter__()` and `__next__()`. The `__iter__()` method must return the instance of the iterator and the `__next__()` method must return the next value from the iterable. These methods are called internally when we use built in functions: `iter()` and `next()`.

Lets create a simple iterable that returns numbers starting from 1:

```
class counter:
    def __iter__(self):
        self.count = 0
        return self
    def __next__(self):
        self.count += 1
        return self.count

c = counter() # Now c is an iterable

c_iter = iter(c)  # c_iter is an iterable

print(next(c_iter))
print(next(c_iter))
print(next(c_iter))
print(next(c_iter))
```

Output:

```
1
2
3
4
```

In the above code:

1. `counter` is a class that is our iterable class.
2. The `__iter__()` method initializes the `count` variable and returns the current instance.
3. The `__next__()` method increases and returns the `count` variable.

This is how we can implement a simple iterable.

Using iterable with for loop:

Just like list, set etc. we can use our custom iterable class with `for` loop. Lets see how:

```
class counter:
    def __iter__(self):
        self.count = 0
        return self
    def __next__(self):
        self.count += 1
        return self.count

c = counter()
```

```
10
11 for i in c:
12     print(i)
```

Output:

```
1
2
3
...
```

But the above program never terminates. The reason is that we never gave any signal or condition to our class about when to stop. But we have a solution for this too.

StopIteration:

In the last program, we could not stop our program when used with `for` loop. We can conditionally stop the iteration by raising `StopIteration` exception. Lets understand by an example:

```
class counter:
    def __iter__(self):
        self.count = 0
        return self
    def __next__(self):
        self.count += 1
        if self.count == 10:
            raise StopIteration
        return self.count

c = counter()

for i in c:
    print(i)
```

Output:

```
1
2
3
4
5
6
7
8
9
```

We have just added a condition in `__next__()` method that when `count` becomes `10`, it should raise `StopIteration`. We have used the `raise` keyword to raise the error.

Implementing Fibonacci series:

A fibonacci series is a popular series that goes on like `0, 1, 1, 2, 3, 5. . .`. The initial two elements are fixed but rest elements are summation of previous two. Ideally we can generate a fibonacci series using this formula:

```
0 = 0
1 = 1
1 = 0 + 1
2 = 1 + 1
3 = 1 + 2
5 = 2 + 3
```

Lets implement this series using iterators:

```
class fib:
    a = 0
    b = 1
    current = -1

    def __init__(self, n):
        self.n = n
    def __iter__(self):
        return self

    def __next__(self):
        self.current +=1
        if self.current == self.n:
            raise StopIteration
        if self.current in [0,1]:
            return self.current
        c = self.a + self.b
        self.a, self.b = self.b, c
        return c

f = fib(10)

for i in f:
    print(i)

```

Output:

```
0
1
1
2
3
5
8
13
21
34
```

Summary:

1. Iterables are containers that store values.
2. Iterators are pointers to the values that can be used to iterate over them.
3. We can create our own iterables using `__iter__()` and `__next__()` methods.
4. To stop an iteration in `for` loop, we must use `StopIteration` error to notify that iteration should be stopped here.

Chapter 21: Generators in Python

Generators are a powerful concept in Python for creating iterators. They allow you to create iterators in a more concise and memory-efficient way. In this chapter, let's explore generators and see how to use them in our applications.

Generator Functions:

A generator function is a special type of function that contains one or more `yield` statements. When a generator function is called, it returns a generator object. This object can be used to iterate over the values produced by the `yield` statements.

Let's create a simple generator function that generates the first five even numbers:

```
def even_numbers():
    yield 2
    yield 4
    yield 6
    yield 8
    yield 10

even_gen = even_numbers()
print(next(even_gen))
print(next(even_gen))
print(next(even_gen))
print(next(even_gen))
print(next(even_gen))
```

Output:

```
2
4
6
8
10
```

In the above code:

- `even_numbers` is a generator function that yields even numbers.
- `even_gen` is the generator object returned by calling `even_numbers()`.
- We use the `next()` function to get the values one by one from the generator.

We can also use a for loop, just like `iterators`

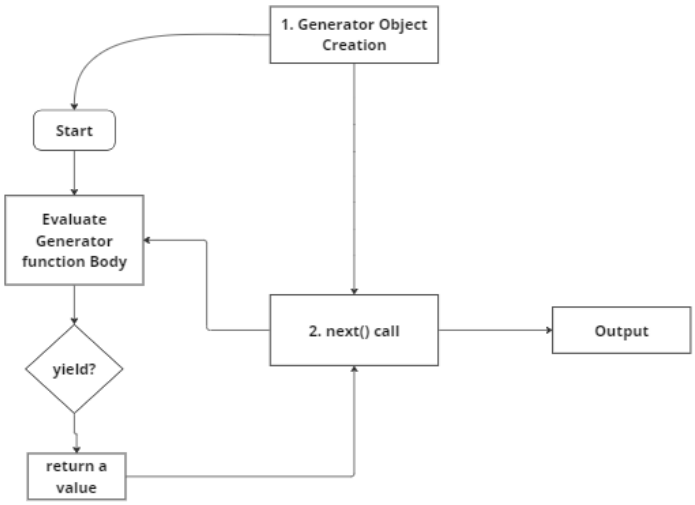
```
def even_numbers():
    yield 2
    yield 4
    yield 6
    yield 8
    yield 10

for num in even_numbers():
    print(num)
```

Output:

```
2
4
6
8
10
```

Note that after function call halts when `next()` is called and resumes from the same line in the next call.

The state of the generator object is remembered and on next call of `next()` function, the function resumes and again halts on next encounter of `yield` statement.



Generator Expressions:

Generator expressions provide a concise way to create generators. They are similar to list comprehensions but use parentheses `()` instead of square brackets `[]`. Generator expressions are memory-efficient because they generate values on-the-fly as you iterate over them.

Here's an example that generates the squares of numbers from 1 to 5 using a generator expression:

```
squares_gen = (x**2 for x in range(1, 6))

for square in squares_gen:
    print(square)
```

Output:

```
1
4
9
16
25
```

In the above code:

- `squares_gen` is the generator expression.
- The syntax is same as list comprehension, the only difference is we used `()` instead of `[]`.

Generator vs. List:

One key difference between generators and lists is that generators do not store all the values in memory at once. This makes generators suitable for working with large datasets or infinite sequences.

Lists, on the other hand, store all values in memory, which can lead to memory issues for large datasets.

Infinite Sequences:

Generators are excellent for working with infinite sequences. Since they produce values on-the-fly, you can create generators for sequences that go on forever, such as the natural numbers:

```
def natural_numbers():
    n = 1
    while True:
        yield n
        n += 1

nat_gen = natural_numbers()
for i in range(5):
    print(next(nat_gen))
```

Output:

```
1
2
3
4
5
```

range() function:

The `range()` function that we often use with `for` loop is also a generator. This is how Python saves memory while using `for` loop, as it doesn't need to generate the whole list of numbers in memory.

We can implement our own custom range function like below:

```
def my_range(start = None, end = None, step = 1):

    if end is None:
        start, end = 0, start
    current = start
    while True:
        yield current
        current += step
        if current >= end:
            break

for i in my_range(4):
    print(i)

```

Output:

```
0
1
```

```
2
3
```

Summary:

- Generators are a versatile tool for working with iterators in Python.
- They provide a memory-efficient way to generate values on-the-fly, making them suitable for various tasks, including handling large datasets and working with infinite sequences.
- When you need to create iterators, consider using generators for a more elegant and efficient solution.

Chapter 22: Decorators in Python

Decorators

Decorators are a powerful and flexible feature in Python that allow you to modify or enhance the behavior of functions or methods without changing their source code. In this chapter, we'll explore decorators and learn how to use them to add functionality to your code.

Introduction to Decorators

A decorator is a higher-order function that takes another function as input and extends its functionality without modifying it directly. Decorators are commonly used for tasks such as logging, authentication, and performance measurement.

Basics of Decorators

To understand decorators, we need to understand, how do we achieve a decorator's functionality without using decorator.

Consider below simple function:

```
def print_value(value):
    print(value)
```

This code does nothing but prints the value we passed. Now, we want to modify this function, without actually editing the body, such as some logging. We can simply add logs inside the function, but that does not achieve our goal to use decorator. Lets create one more function that takes our function and returns another function with additional logging:

```
def logging(func):
    def wrapper(value):
        print("Before calling function")
        func(value)
        print("After calling function")
    return wrapper

def print_value(value):
    print(value)
```

In the above code,

- `logging` is a new function that takes our function as input.
- `wrapper` is an inner function that is having same signature as our `print_value` function.
- Inside `wrapper`, we are implementing our "additional" functionalities and also calling our `print_value` function
- Note that we didn't modify our original function yet.

Now how do we use it? Its simple, just call the logging function and pass our original function like below:

```
def logging(func):
    def wrapper(value):
        print("Before calling function")
        func(value)
        print("After calling function")
    return wrapper

def print_value(value):
    print(value)
```

```
10
11 new_print_value = logging(print_value)  # Returns a new function
12 new_print_value("Hello, world!")
```

Output:

```
1 Before calling function
2 Hello, world!
3 After calling function
```

This is the basic working of decorator. But this is not decorator yet. Python allows us to use a shorter syntax to avoid creating the new function every time. We can use `@logging` above our function to modify its functionality directly:

```
1  def logging(func):
2      def wrapper(value):
3          print("Before calling function")
4          func(value)
5          print("After calling function")
6      return wrapper
7
8  @logging
9  def print_value(value):
10     print(value)
11
12 print_value("Hello, world!")
```

Output:

```
1 Before calling function
2 Hello, world!
3 After calling function
```

Note that we are no longer creating a new function, we can directly call our original function and still get the additional functionalities.

In a nut shell, below is the syntax of decorators:

```
def decorator(function):
    def wrapper(*args, **kwargs):
        # do the additional stuff
        function(*args, **kwargs)  # call the original function
    return wrapper

@decorator
def my_function(*args, **kwargs):
    # function body
```

- Pass a function in the decorator function
- Define an inner function and do the additional stuff inside it
- Call the original function from inner function
- The signature of inner function should be same as original function
- Return the internal function from decorator function

- Use `@decorator_name` above original function to apply the decorator.

Lets see some examples of decorators:

Example 1: Timing Decorator

```
import time

def timing_decorator(func):

    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Time taken by {func.__name__}: {end_time - start_time} seconds")
        return result
    return wrapper

@timing_decorator
def some_function():
    time.sleep(2)
    print("Function executed")

some_function()
```

Output:

```
Function executed
Time taken by some_function: 2.00 seconds
```

Explanation:

In this example, the `timing_decorator` is applied to the `some_function`. The decorator calculates and prints the time taken by the function to execute, including the sleep time.

Example 2: Logging Decorator

```
def logging_decorator(func):

    def wrapper(*args, **kwargs):
        print(f"Calling function {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} executed")
        return result
    return wrapper

@logging_decorator
def greet(name):
    print(f"Hello, {name}")
greet("Alice")
```

Output:

```
Calling function greet
Hello, Alice
```

```
Function greet executed
```

Explanation:

In this example, the `logging_decorator` is used to wrap the `greet` function. The decorator prints messages before and after the function execution to log its calling and completion.

Decorating Functions with Arguments:

If the original function takes arguments, the decorator can be modified to handle those arguments.

Example 1: Argument Decorator

```
def argument_decorator(func):

    def wrapper(*args, **kwargs):
        print("Arguments:", args)
        result = func(*args, **kwargs)
        return result
    return wrapper

@argument_decorator

def add(a, b):
    return a b
add(3, 5)
```

Output:

```
Arguments: (3, 5)
```

Explanation:

The `argument_decorator` is applied to the `add` function. The decorator prints the arguments passed to the function before executing it.

Example 2: Argument Validation Decorator

```
def validate_arguments(func):

    def wrapper(*args, **kwargs):
        for arg in args:
            if not isinstance(arg, (int, float)):
                raise ValueError("Arguments must be numbers")
        result = func(*args, **kwargs)
        return result
    return wrapper

@validate_arguments

def multiply(x, y):
    return x * y
multiply(4, 7.5)
```

Output:

```
30.0
```

Explanation:

The `validate_arguments` decorator checks if the arguments are numbers before executing the decorated function. If any argument is not a number, it raises a `ValueError`.

Chaining Decorators:

You can apply multiple decorators to a single function.

Example 1: Multiple Decorators

```
def decorator1(func):
    def wrapper():
        print("Decorator 1")
        func()
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2")
        func()
    return wrapper

@decorator1
@decorator2
def my_function():
    print("Original function")
my_function()
```

Output:

```
Decorator 1
Decorator 2
Original function
```

Explanation:

The `my_function` is passed through both `decorator2` and `decorator1`. The order of decoration determines the order of execution of decorators.

Example 2: Order of Decoration Matters

```
@decorator2
@decorator1
def another_function():
    print("Another function")
another_function()
```

Output:

```
Decorator 2
Decorator 1
Another function
```

Explanation:

In this case, `another_function` is first passed through `decorator1` and then through `decorator2`, causing the decorators to execute in reverse order.

Summary:

- Decorators extend the functionality of functions without modifying their source code.
- Decorators are applied using the `@decorator_name` syntax.
- Decorators can be used for tasks like timing, logging, and argument validation.
- Chaining decorators allows you to apply multiple decorators to a single function.

Chapter 23: Regular Expressions in Python

Regular expressions

Regular expressions, often referred to as regex, are powerful tools for pattern matching and text manipulation in Python. They allow you to search for, match, and manipulate strings based on specific patterns. In this chapter, we'll explore regular expressions and how to use them effectively in Python applications.

Introduction to Regular Expressions

Regular expressions are sequences of characters that define a search pattern. They are used for pattern matching within strings.

Basic Patterns and Characters

- `.` : Matches any character except a newline.
- `\d` : Matches any digit (0-9).
- `\w` : Matches any word character (a-z, A-Z, 0-9, _).
- `\s` : Matches any whitespace character.
- `[]` : Defines a character set. For example, `[aeiou]` matches any vowel.
- `[^]` : Matches any character NOT in the set.
- `|` : Acts like an OR operator.

Quantifiers

- `*` : Matches 0 or more occurrences of the preceding character.
- `+` : Matches 1 or more occurrences of the preceding character.
- `?` : Matches 0 or 1 occurrence of the preceding character.
- `{n}` : Matches exactly n occurrences of the preceding character.
- `{n,}` : Matches n or more occurrences of the preceding character.
- `{n,m}` : Matches between n and m occurrences of the preceding character.

Anchors

- `^` : Matches the start of a string.
- `$` : Matches the end of a string.

Using Regular Expressions in Python

To use regular expressions in Python, we need the `re` module.

1. re.match()

The `re.match()` function searches for a pattern only at the beginning of the string.

Example:

Suppose we need to find whether the given string starts with a given pattern or not. It can be useful in case we need to put some validations on API inputs.

```
import re
pattern = r"\d{3}"
text = "123abc456"
match = re.match(pattern, text)
if match:
    print("Match found:", match.group())
else:
    print("No match")
```

Output:

```
Match found: 123
```

Explanation:

In this example, the `re.match()` function attempts to find the pattern `r"\d{3}"` (which matches exactly three digits) at the beginning of the string `text`. Since the pattern "123" is found at the beginning of the string, the match is successful, and the `match.group()` method retrieves the matched substring "123".

2. re.search()

The `re.search()` function searches for a pattern anywhere in the string.

Example:

Suppose we want to validate a string if it contains a certain pattern or not, we can use `search()` function.

```
pattern = r"\d{3}"
text = "abc123def"

search_result = re.search(pattern, text)

if search_result:
    print("Match found:", search_result.group())
else:
    print("No match")
```

Output:

```
Match found: 123
```

Explanation:

This example uses the `re.search()` function to locate the pattern `r"\d{3}"` within the string `text`. The pattern matches the substring "123", and since it's found within the text, the search result is successful. The `search_result.group()` method retrieves the matched substring "123".

3. re.findall()

The `re.findall()` function returns a list of all occurrences of a pattern in the string.

Example:

```
pattern = r"\d "
text = "There are 123 apples and 456 oranges."

matches = re.findall(pattern, text)

print("Matches:", matches)
```

Output:

```
Matches: ['123', '456']
```

Explanation:

Here, the `re.findall()` function looks for all instances of the pattern `r"\d "`, which matches one or more digits, in the string `text`. The result is a list containing the matched substrings "123" and "456".

4. re.finditer()

The `re.finditer()` function returns an iterator yielding match objects for all occurrences of a pattern.

Example:

```
pattern = r"\d "

text = "There are 123 apples and 456 oranges."
match_iter = re.finditer(pattern, text)

for match in match_iter:
    print("Match found:", match.group())
```

Output:

```
Match found: 123
Match found: 456
```

Explanation:

Using `re.finditer()`, an iterator is created that yields match objects for all occurrences of the pattern `r"\d "` in the text. The loop iterates through each match object, and `match.group()` retrieves the matched substring. In this case, the output demonstrates the matches "123" and "456".

5. re.sub()

The `re.sub()` function replaces occurrences of a pattern with a replacement string.

Example:

```
pattern = r"\d "
text = "There are 123 apples and 456 oranges."

replacement = "X"

new_text = re.sub(pattern, replacement, text)
```

```
7
8 print("Original:", text)
9 print("Modified:", new_text)
```

Output:

```
1 Original: There are 123 apples and 456 oranges.
2 Modified: There are X apples and X oranges.
```

Explanation:

The `re.sub()` function searches for all instances of the pattern `r"\d "` in the text and replaces them with the string "X". The output shows both the original text and the modified text, where the digits "123" and "456" have been replaced with "X".

6. re.split()

The `re.split()` function splits a string by occurrences of a pattern.

Example:

```
1 pattern = r"\s "
2 text = "Hello   world  !"
3
4 parts = re.split(pattern, text)
5
6 print("Parts:", parts)
```

Output:

```
1 Parts: ['Hello', 'world', '!']
```

Explanation:

The `re.split()` function divides the string `text` into parts using the pattern `r"\s "`, which matches one or more whitespace characters. As a result, the output displays a list of parts obtained from splitting the text at spaces.

Summary

- Regular expressions are patterns used for text manipulation.
- Special characters like `.` `\d`, `\w`, and `\s` have specific meanings.
- Quantifiers like `*`, `+`, `?`, `{}`, and anchors `^`, `$` refine searches.
- The `re` module is used in Python for regular expressions.
- `re.match()` searches for a pattern at the start of a string.
- `re.search()` searches for a pattern anywhere in a string.
- `re.findall()` returns a list of all occurrences of a pattern.
- `re.finditer()` returns an iterator yielding match objects.
- `re.sub()` replaces occurrences of a pattern with a replacement.
- `re.split()` splits a string by occurrences of a pattern.

Chapter 24: Math Operations in Python

Basic Math Operations in Python

In this chapter, we'll explore basic math operations in Python and see how to use them in our applications.

Addition, Subtraction, Multiplication, and Division

Python provides simple and intuitive ways to perform basic arithmetic operations.

Addition (+)

Example:

```
result = 5+3
print(result)
```

Output:

```
8
```

Subtraction (-)

```
result = 10 - 4
print(result)
```

Output:

```
6
```

Multiplication (*)

```
result = 7 * 2

print(result)
```

Output:

```
14
```

Division (/)

```
result = 15 / 3

print(result)
```

Output:

```
5.0
```

```
1
```

Note: Division always returns a float

Exponents and Modulus

Exponents (**)

You can calculate exponentiation in Python using the double asterisk `**`.

```
result = 2 ** 3

print(result)
```

Output:

```
8
```

Modulus (%)

The modulus operator `%` returns the remainder of a division.

```
result = 17 % 5
print(result)
```

Output:

```
2
```

Note: 17 divided by 5 leaves a remainder of 2

Order of Operations

Python follows the standard order of operations (PEMDAS: Parentheses, Exponents, Multiplication and Division, Addition and Subtraction).

```
result = 3 + 4 * 2

print(result)
```

Output:

```
11
```

Note: Multiplication is done before addition

Math Functions

Python provides several built-in math functions in the `math` module.

```
import math
# Square root
result = math.sqrt(25)
print(result)
# Absolute value
```

```
result = abs(-7)
print(result)
# Round to the nearest integer
result = round(6.75)
print(result)
```

Output

```
5.0
7
7
```

Advanced Math Concepts in Python

Now we'll dive deeper into advanced math concepts in Python and explore more complex mathematical operations.

1. Trigonometric Functions

Python's `math` module provides a wide range of trigonometric functions.

Sine, Cosine, and Tangent

```
import math

angle = math.pi / 6  # 30 degrees in radians

sin_result = math.sin(angle)

cos_result = math.cos(angle)

tan_result = math.tan(angle)

print(sin_result)

print(cos_result)

print(tan_result)
```

Output:

```
0.5
0.86602540378
0.57735026919
```

2. Logarithms

You can calculate logarithms using the `math` module.

Natural Logarithm (ln)

```
import math

result = math.log(2.71828)  # Natural logarithm of e
```

```
4
5 print(result)
```

Output:

```
1 1.0
```

Base-10 Logarithm (log10)

```
1 import math
2
3 result = math.log10(100)
4
5 print(result)
```

Output:

```
1 2.0
```

3. Constants

Python's `math` module provides access to important mathematical constants.

Pi (π)

```
1 import math
2
3 pi_value = math.pi
4
5 print(pi_value)
```

Output:

```
1 3.141592653589793
```

4. Random Numbers

Python's `random` module allows you to generate random numbers for various applications, including simulations and games.

Generating Random Integers

```
1 import random
2
3 random_integer = random.randint(1, 100)
4
5 print(random_integer)
```

Output:

```
1 Random integer between 1 and 100
```

Generating Random Floating-Point Numbers

```
import random

random_float = random.uniform(0, 1)

print(random_float)
```

Output:

```
Random float between 0 and 1
```

5. Complex Numbers

Python supports complex numbers and provides functions for complex arithmetic. A complex number is an extension to real numbers where we deal with imaginary numbers `√-1`. In Mathematics, we denote it with symbol `i` but in python we denote it using `j` after the number.

```
complex_num_1 = 3 + 4j

complex_num_2 = 2 - 2j

result = complex_num_1 + complex_num_2

print(result)
```

Output:

```
(5 2j)
```

6. Numerical Integration and Differentiation

For advanced mathematical computations, you can use libraries like NumPy and SciPy to perform numerical integration and differentiation. NumPy and SciPy are complex libraries written in C++ to support fast computation. These libraries are the base of machine learning programs in Python which make the programming of AI very user friendly and fast.

```
import numpy as np
from scipy.integrate import quad
from scipy.misc import derivative
# Define a function for integration and differentiation
def f(x):
    return x**2

# Integration
integral, _ = quad(f, 0, 1)
print(integral)

# Differentiation
derivative_result = derivative(f, 2.0, dx=1e-6)
print(derivative_result)
```

Output:

```
0.3333333333333333
4.000000000061299
```

Summary

In this chapter, we explored basic math operations in Python, including addition, subtraction, multiplication, division, exponentiation, and modulus. We also learned about the order of operations and how to use some common math functions from the `math` module.

These fundamental math concepts are essential for various mathematical calculations and are the building blocks for more advanced mathematical operations in Python.

We delved into advanced math concepts in Python, including trigonometric functions, logarithms, mathematical constants, random number generation, complex numbers, and even numerical integration and differentiation using libraries like NumPy and SciPy.

These advanced mathematical tools open up a wide range of possibilities for scientific computing, data analysis, and more complex mathematical modeling in Python.

Chapter 25: File Handling in Python

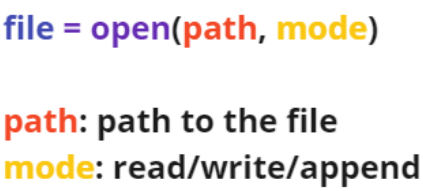
File Handling in Python

File handling is an essential aspect of programming, allowing us to interact with files on our computer. Whenever we execute a program, the data persists in our memory, which is lost as soon as the program is finished executing. Sometimes, we need to store some useful data in the hard drive that was computed by our program. There are two basic approaches, we either store it in a file, or in a database. Lets focus on files in this chapter. In Python, you can easily work with files to read, write and manipulate data. In this tutorial, we will explore file handling in Python, step by step, with code examples.

Opening a File

Before you can perform any operations on a file, you need to open it. Python provides the `open()` function for this purpose. Let's start by opening a file in read-only mode:

Syntax:




```
# Opening a file in read-only mode
file = open("example.txt", "r")
```

In the code above:

- `"example.txt"` is the name of the file we want to open.
- `"r"` specifies that we are opening the file in read-only mode.
- `open()` will return a `File` object.
- Note that `open` function takes the path to the file.
  - It can be a full path : `C:\Users\Documents\file.txt`
  - or, A relative path : `src\app\data.txt`
  - In case of relative path, the root of the path is the same location where our main program is located.

Reading from a File

Once the file is open, you can read its contents. Let's read the entire file and print its content:

```
# Reading the entire file
file = open("example.txt", "r")
content = file.read()
print(content)
```

Here, `file.read()` reads the entire content of the file, and `print(content)` displays it. Note that `read()` is a method of `File` class.

Closing a File

After you're done with a file, it's crucial to close it using the `close()` method to free up system resources:

```
# Reading the entire file
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

Failure to close a file can lead to resource leaks and potential issues when working with the file later. It is a good programming practice to release a resource after using it. The reason is, some other process might want to use it and we are unnecessarily holding the resource, which may lead to a deadlock.

Writing to a File

Now, let's learn how to write data to a file. To write to a file, you need to open it in write mode (`"w"`). If the file doesn't exist, Python will create it. Be cautious when using write mode, as it will overwrite the file if it already exists. Overwriting means, it will clear the content and write from the beginning.

```
# Opening a file in write mode
file = open("output.txt", "w")

# Writing data to the file
file.write("Hello, World!\n")

file.write("This is a new line of text.")

# Closing the file
file.close()
```

In this example, we open the file `"output.txt"` in write mode, write two lines of text, and then close the file. If we go to the file explorer and open the file, we will see the content we have written through our program.

Appending to a File

With `write()` method, there is a problem that we can not continue writing once the file is closed. To add content to an existing file without overwriting it, open the file in append mode (`"a"`):

```
# Opening a file in append mode
file = open("output.txt", "a")

# Appending data to the file
file.write("\nThis line is appended.")

# Closing the file
file.close()
```

Appending mode allows you to add content to the end of the file without erasing the previous content.

Using the `with` Statement

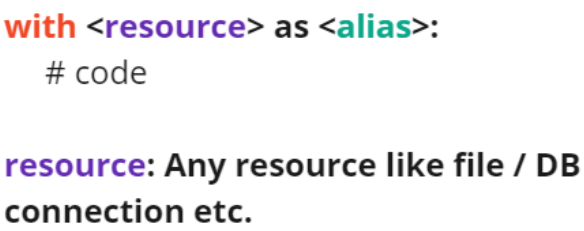
Python provides a convenient way to open and close files automatically using the `with` statement. This ensures that the file is properly closed, even if an exception occurs:

```
# Using the with statement
with open("example.txt", "r") as file:
```

```
    content = file.read()
    print(content)

# File is automatically closed when exiting the 'with' block
```

Note that `with` statement is not specific to file handling. Any resource that needs to be closed, can be loaded using with operator to get it automatically closed. In Python, such functions or classes are known as `Context Manager`.

Syntax:



Reading Line by Line

When working with large files, it's often more efficient to read them line by line. You can achieve this using a `for` loop:

```
# Reading a file line by line

with open("example.txt", "r") as file:
    for line in file:
        print(line, end="")
```

This method is memory-efficient because it doesn't load the entire file into memory.

Checking if a File Exists

Before opening a file, it's a good practice to check if it exists to avoid errors. We can use the `os` module for this. Python's `os` module provides many features to working with Operating System, and file handling is one of those features.

```
import os

if os.path.exists("example.txt"):
    # File exists, proceed with opening it
    with open("example.txt", "r") as file:
        content = file.read()
        print(content)
else:
    print("File does not exist.")
```

In above example,

- `os.path.exists()` is a built in function to check if the given file is present or not.
- If it exists, we are reading the file and printing the content

Conclusion

In this tutorial, we covered the basics of file handling in Python, including opening, reading, writing, and appending to files. We also discussed the use of the `with` statement for proper file handling and checked for the existence of files. File handling is a crucial skill for any Python programmer, as it enables interaction with external data sources and file-based storage.

Chapter 26: Exception handling in Python

Exception handling:

Exception handling is a fundamental aspect of programming that allows us to gracefully handle errors and unexpected situations that may arise during the execution of a program. In Python, exceptions are raised when an error occurs, and these exceptions can be caught and handled using exception handling mechanisms. In this tutorial, we will explore exception handling in Python, step by step, with code examples and explanations.

What are Exceptions?

An exception is an event that occurs during the execution of a program, disrupting the normal flow of the program's instructions. Exceptions can be caused by various factors, such as incorrect input, division by zero, or attempting to access a file that doesn't exist. When an exception is raised, Python generates an exception object containing information about the error.

Note that an Exception is one of those situations where our code doesn’t work for some specific set of inputs. We need to have a mechanism to handle it so that our program doesn’t break.

Error vs Exceptions

An Error is, a situation when our program does not work at all. It can be due to wrong syntax or wrong design / logic of writing the program.

But an Exception is a situation when our program works, but it fails in some specific scenarios.

Handling Exceptions

Python provides a structured way to handle exceptions using the `try` and `except` blocks. The `try` block contains the code that might raise an exception, and the `except` block contains the code to handle the exception if it occurs.

Let's start with a simple example:

```
# Handling a ZeroDivisionError

try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("An error occurred:", e)
```

Output:

```
An error occurred: division by zero
```

In this example, we attempt to divide by zero, which raises a `ZeroDivisionError` exception. The `except` block catches the exception, and we can access the error message using `as e` and print a custom error message.

Handling Multiple Exceptions

You can handle multiple exceptions by using multiple `except` blocks. This allows you to provide specific error handling for different types of exceptions.

```
# Handling multiple exceptions

try:
    result = 10 / 0
```

```
except ZeroDivisionError as e:
    print("Division by zero:", e)
except ValueError as e:
    print("Value error:", e)
except Exception as e:
    print("An error occurred:", e)
```

Output:

```
An error occurred: division by zero
```

In this example, we catch a `ZeroDivisionError` and a `ValueError` separately, and then a general `Exception` to handle any other exceptions.

If the raised error is not matched, it will move to the next `except` block until it finds a match. We generally use an `Exception` block to ensure our error is caught, as `Exception` class is the parent of all the classes.

The `else` and `finally` Blocks

In addition to `try` and `except`, Python also provides the `else` and `finally` blocks for more advanced exception handling:

- The `else` block is executed if no exceptions are raised in the `try` block. It is useful for code that should run only when there are no exceptions.

```
# Using the else block

try:
    result = 10 / 2
except ZeroDivisionError as e:
    print("Division by zero:", e)
else:
    print("No exceptions occurred. Result:", result)
```

Output:

```
No exceptions occurred. Result: 5.0
```

- The `finally` block is executed regardless of whether an exception was raised or not. It is commonly used for cleanup operations like closing files or network connections.

```
# Using the finally block
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError as e:
    print("File not found:", e)
finally:
    print("Inside finally block: closing file")
    if 'file' in locals():
        file.close()
```

Output:

```
Inside finally block: closing file
```

Custom Exceptions

Python allows you to create custom exceptions by defining new classes that inherit from the built-in `Exception` class. Custom exceptions can provide meaningful error messages and enhance the clarity of your code.

```
# Creating a custom exception
class CustomError(Exception):
    def __init__(self, message):
        self.message = message

try:
    raise CustomError("This is a custom exception")
except CustomError as e:
    print("Custom exception:", e.message)
```

Output:

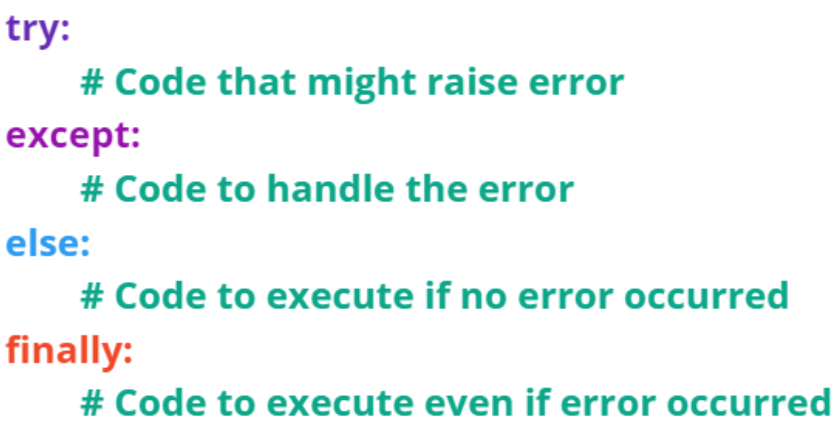
```
Custom exception: This is a custom exception
```

In this example,

- We created a custom class that extends `Exception` class.
- In the `__init__` method, we are setting the `message` variable, which will set the error message for us.
- When we use `raise` keyword, it will disrupt the program and we can handle it to customize our program for specific inputs.

Order of Execution:

Below image describes the order of keywords while handling exception.



Note that `try` is never used alone. It must be used with `except`, `finally` or both.

Conclusion

Exception handling is a crucial part of Python programming that helps you write robust and error-tolerant code. By using `try`, `except`, `else`, and `finally` blocks, you can gracefully handle errors and exceptions, making your programs more reliable and user-friendly. Custom exceptions allow you to provide specific error messages tailored to your application's needs.

Chapter 27: Date & Time in Python

Introduction:

Working with dates and times is a common requirement in many programming tasks. Python provides a powerful module called `datetime` that allows you to work with dates, times, and time intervals. In this tutorial, we will explore how to work with date and time in Python using the `datetime` module.

The `datetime` Module

The `datetime` module in Python provides classes for working with dates and times. It includes several classes such as `datetime`, `date`, `time`, `timedelta`, and more, each serving a specific purpose in date and time manipulation.

Let's start by importing the `datetime` module:

```
import datetime
```

Current Date and Time

You can obtain the current date and time using the `datetime.now()` function:

```
# Current date and time
import datetime
current_datetime = datetime.datetime.now()
print("Current date and time:", current_datetime)
```

Output:

```
Current date and time: 2023-10-02 18:27:40.149255
```

In this code, `datetime.datetime.now()` returns the current date and time, which is stored in the `current_datetime` variable and then printed.

Date and Time Components

The `datetime` object has attributes to access its individual components, such as year, month, day, hour, minute, second, and microsecond:

```
import datetime

current_datetime = datetime.datetime.now()
# Accessing date and time components
year = current_datetime.year
month = current_datetime.month
day = current_datetime.day
hour = current_datetime.hour
minute = current_datetime.minute
second = current_datetime.second
microsecond = current_datetime.microsecond

print("Year:", year)
print("Month:", month)
print("Day:", day)
print("Hour:", hour)
print("Minute:", minute)
```

```
print("Second:", second)
print("Microsecond:", microsecond)
```

Output:

```
Year: 2023
Month: 10
Day: 2
Hour: 18
Minute: 19
Second: 10
Microsecond: 509010
```

In this example, we extract and print various components of the current date and time.

Formatting Dates as Strings

You can format dates as strings using the `strftime()` method. This method allows you to specify the desired format using format codes:

```
import datetime

current_datetime = datetime.datetime.now()

# Formatting dates as strings
formatted_date = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted date:", formatted_date)
```

Output:

```
Formatted date: 2023-10-02 18:20:19
```

Here, `strftime("%Y-%m-%d %H:%M:%S")` formats the date and time as "YYYY-MM-DD HH:MM:SS."

Parsing Strings to Dates

You can convert date strings into `datetime` objects using the `strptime()` method:

```
import datetime

# Parsing strings to dates
date_string = "2023-10-02 15:30:00"
parsed_date = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print("Parsed date:", parsed_date)
```

Output:

```
Parsed date: 2023-10-02 15:30:00
```

Here, `strptime(date_string, "%Y-%m-%d %H:%M:%S")` parses the date string using the specified format.

Date Arithmetic

Python's `datetime` module allows you to perform arithmetic operations on dates and times. You can calculate the difference between two dates using the `timedelta` class:

```
# Date arithmetic with timedelta
import datetime
from datetime import timedelta
```

```
4
5 date1 = datetime.date(2023, 10, 2)
6 date2 = datetime.date(2023, 10, 5)
7
8 difference = date2 - date1
9 print("Difference in days:", difference.days)
```

Output:

```
1 Difference in days: 3
```

In this example, we calculate the difference in days between `date1` and `date2` using `timedelta`.

Adding and Subtracting Time Intervals

You can add or subtract time intervals to/from dates using `timedelta`:

```
1  # Adding and subtracting time intervals
2  import datetime
3  from datetime import timedelta
4
5  current_date = datetime.date(2023, 10, 2)
6  one_week_later = current_date + timedelta(weeks=1)
7  three_days_ago = current_date - timedelta(days=3)
8
9  print("One week later:", one_week_later)
10 print("Three days ago:", three_days_ago)
```

Output:

```
1 One week later: 2023-10-09
2 Three days ago: 2023-09-29
```

Here, we calculate `one_week_later` and `three_days_ago` by adding and subtracting time intervals from `current_date`.

Conclusion

In this tutorial, we've explored date handling in Python using the `datetime` module. We covered obtaining the current date and time, accessing date and time components, formatting dates as strings, parsing strings into dates, performing date arithmetic with `timedelta`, and adding/subtracting time intervals. Date handling is essential for various applications, including scheduling, data analysis, and more.

Chapter 28: Multithreading in Python

Multithreading in Python:

Multithreading is a technique in Python that allows you to execute multiple threads (smaller units of a process) concurrently, enabling better utilization of CPU resources and improved program responsiveness. In this tutorial, we'll explore multithreading in Python step by step.

What is Threading?

Threading is a programming technique where a single process splits into multiple threads, each capable of executing code independently. Threads share the same memory space, allowing them to interact and communicate more efficiently than separate processes. This concurrency enables improved CPU utilization and responsiveness in applications.

We can run multiple threads in parallel, without creating a separate process. This concept is known as multithreading.

Threads vs. Processes

Threads and processes are both concurrency mechanisms, but they differ in several key aspects.

Threads	Processes
• Threads share the same memory space and resources. • Threads are lightweight compared to processes and have lower overhead. • Threads are suitable for I/O-bound tasks and tasks requiring shared memory. • Threads are affected by the GIL, limiting true parallelism for CPU-bound tasks.	• Processes have separate memory spaces and resources. • Processes have higher overhead and consume more memory. • Processes are ideal for CPU-bound tasks that benefit from true parallelism. • Processes are not affected by the GIL, allowing better CPU utilization.

Creating and Using Threads:

To work with threads in Python, you need to import the `threading` module.

```
import threading
```

Creating and Running Threads

Let's start with a simple example of creating and running threads. To create a thread object, we need to use the `Thread` class of `threading` module.

```
import threading

def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")

def print_letters():
    for letter in 'abcde':
        print(f"Letter {letter}")

# Create two thread objects
```

```
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

# Start the threads
t1.start()
t2.start()

# Wait for both threads to finish
t1.join()
t2.join()
```

Output:

```
Number 1
Letter a
Number 2
Letter b
Number 3
Letter c
Number 4
Letter d
Number 5
Letter e
```

In this example,

- We defined two functions, `print_numbers` and `print_letters`, and created two thread objects `t1` and `t2`.
- While creating thread object, we pass the target function to be executed in the `target` parameter.
- We then started both threads using `start` method and waited for them to finish using the `join` method.
- We can easily observe that the functions are executing in parallel as the output is mixed of both functions.
- One important point to note is, the output will be different, on different machines or if we rerun the same program.

Passing Arguments to Threads

We can also pass arguments to threads when creating them in the `args` parameter.

```
import threading

def print_numbers(num):
    for i in range(1, num 1):
        print(f"Number {i}")

def print_letters():
    for letter in 'abcde':
        print(f"Letter {letter}")

t1 = threading.Thread(target=print_numbers, args=(3,))
t2 = threading.Thread(target=print_letters)

t1.start()
t2.start()

t1.join()
t2.join()
```

Output:

```
Number 1
Number 2
Number 3
Letter a
Letter b
Letter c
Letter d
Letter e
```

Here, we passed an argument `3` to the `print_numbers` thread using the `args` parameter.

Race Condition:

When two process or threads are running in parallel and they use the same object for some computation purpose, it might happen, that due to a dirty write, the threads incorrectly update the state of that object. This is called race condition.

A simple example is, suppose we have two parallel transactions on below code:

```
function increase_balance(amount):

    1. read total_balance in local_balance

    2. local_balance = local_balance + amount

    3. write local_balance in total_balance
```

Lets say total_balance=100 and amount = 5. If there are two threads, calling the same function, in case of a dirty write, we might execute this function in below order:

```
Thread 1 : read total_balance in local_balance (local_balance = 100)

Thread 2 : read total_balance in local_balance (local_balance = 100)

Thread 1: local_balance = local_balance + amount (local_balance = 100 + 5 = 105)

Thread 2: local_balance = local_balance + amount (local_balance = 100 + 5 = 105)

Thread 1: write local_balance in total_balance (total_balance = 105)

Thread 2: write local_balance in total_balance (total_balance = 105)
```

Which is incorrect because it should have been 110, not 105. This is called race condition.

Synchronization with Locks:

In multithreaded programs, it's essential to synchronize threads to avoid race conditions. Python provides the `threading.Lock` class for this purpose.

```
import threading

counter = 0
counter_lock = threading.Lock()

def increment_counter():
    global counter
    with counter_lock:
        for _ in range(1000000):
            counter += 1
```

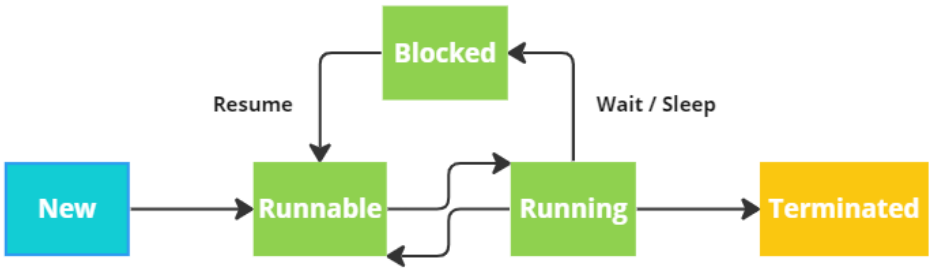
```
11
12 def decrement_counter():
13     global counter
14     with counter_lock:
15         for _ in range(1000000):
16             counter -= 1
17
18 t1 = threading.Thread(target=increment_counter)
19 t2 = threading.Thread(target=decrement_counter)
20
21 t1.start()
22 t2.start()
23
24 t1.join()
25 t2.join()
26
27 print("Counter:", counter)
```

Output:

```
1 Counter: 0
```

In this example, we use a lock (`counter_lock`) to synchronize access to the `counter` variable between two threads. Without the lock, the counter value would be unpredictable due to race conditions.

Thread Lifecycle:



A thread has below states in its lifecycle:

- **New :** When the thread is just created
- **Runnable:** A thread that is ready to be executed but still waiting for CPU or resources
- **Running:** A thread that is in execution phase
- **Blocked:** Once the thread is sleeping or waiting to put lock on a resource, it remains in blocked state. After this state, it resumes the execution and goes in Runnable state
- **Terminated:** Once the execution is over, thread goes in terminated state.

Understanding the Global Interpreter Lock (GIL)

One unique characteristic of Python is the Global Interpreter Lock (GIL). The GIL is a mutex (mutual exclusion) that allows only one thread to execute Python bytecode at a time, even on multi-core CPUs. This may seem counterintuitive for a language designed for simplicity and ease of use.

The GIL was introduced to address issues related to memory management in Python's CPython implementation. It simplifies memory management by ensuring that only one thread accesses Python objects at a time. However, it limits the potential for true parallelism in CPU-bound tasks.

GIL is a problem for Python’s threading performance. But it might be resolved in future releases.

Conclusion:

Multithreading in Python allows you to execute tasks concurrently, improving performance in certain scenarios. However, it requires careful handling of shared resources to avoid issues like race conditions. Python's `threading` module provides the tools necessary to work with threads effectively.

Chapter 29: Package Management using PIP

Python is a powerful programming language with a vast ecosystem of libraries and packages that can enhance your development experience. Managing these packages efficiently is essential for building robust Python applications. Whenever we need to use a 3rd party library that is not present in our system, we need to download it and add into our local repository in order to use it. Also we might have different versions of each library in multiple applications. In this tutorial, we will explore the key components for managing Python packages: `pip` and `virtualenv`.

What Are PIP and VirtualEnv?

- `pip` is a package manager for Python that allows you to install, upgrade, and manage Python packages. It simplifies the process of adding external libraries to your Python environment.
- `virtualenv` is a tool that creates isolated Python environments. It enables you to work on different Python projects with their own dependencies, preventing conflicts between packages.
- `PyPI` is a repository of Python packages and libraries contributed by the Python community. It serves as the central hub for discovering and distributing Python packages.

Installing PIP

Before using `pip`, you need to ensure it is installed. Most modern Python installations come with `pip` pre-installed, but it was not the case in older versions. To check if `pip` is available, open your terminal or command prompt and run:

```
pip --version
```

If `pip` is not installed, you can install it manually by using below command:

```
python -m ensurepip --upgrade
```

Alternatively, we can visit the official page of PIP and follow the latest methods of installing PIP.

Installing virtualenv

To install `virtualenv`, use `pip`:

```
pip install virtualenv
```

Now that we have `pip` and `virtualenv` installed, let's proceed with creating a virtual environment.

Creating a Virtual Environment

A virtual environment isolates your Python project, allowing you to install packages independently. To create a virtual environment, run:

```
# Create a virtual environment named 'myenv'
virtualenv myenv
```

This command creates a directory called `myenv` containing a clean Python environment. To activate the virtual environment:

- On Windows:

```
myenv\Scripts\activate
```

- On macOS and Linux:

```
source myenv/bin/activate
```

Your terminal prompt should now indicate that you are in the virtual environment.

Installing Packages with PIP

Inside your virtual environment, you can use `pip` to install Python packages. For example, to install the `requests` library:

```
pip install requests
```

This command installs the `requests` library in your virtual environment. You can verify by either importing it into a program, or by using interactive mode of Python:

```
C:\>python
Python 3.10.7 (tags/v3.10.7:6cc6b13) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> requests.__version__
'2.31.0'
>>>
```

Why do we need a virtualenv?

If we execute the pip commands without creating a virtualenv, it will still work but it will install all the packages in the global location.

While working on multiple projects from same system, we generally encounter one issue that the common libraries in those projects are having different versions. Sometimes, even the python version is different among the projects. If the versions are different, we can not use global libraries in our system, as it will break one or the other project.

To resolve this issue, it is a good practice to always create a virtualenv and install the libraries in its local space. This way there will never be any conflict of versions and we can develop multiple projects in our single system.

Listing Installed Packages

To list the packages installed in your virtual environment, use:

```
pip list
```

This command displays a list of installed packages along with their versions.

Output:

```
C:\Users\mukul.bindal>pip list
Package            Version
------------------ ---------
autopep8           2.0.2
certifi            2023.7.22
charset-normalizer 3.3.0
distlib            0.3.4
filelock           3.7.1
general-toolkit    0.0.3
idna               3.4
install            1.3.5
...
```

Creating a requirements.txt File

To document the packages used in your project and their versions, you can create a `requirements.txt` file. This file lists all the dependencies, making it easier to recreate your virtual environment on another system. To generate a `requirements.txt` file:

```
pip freeze > requirements.txt
```

This command saves the currently installed packages and their versions to a `requirements.txt` file.

Installing Packages from requirements.txt

To recreate a virtual environment with the same packages listed in a `requirements.txt` file, use:

```
# Create a new virtual environment
virtualenv newenv

# Activate the new environment
source newenv/bin/activate  # On macOS and Linux
newenv\Scripts\activate  # On Windows

# Install packages from requirements.txt
pip install -r requirements.txt
```

This process allows you to easily set up the same environment on different systems.

Conclusion

In this tutorial, we've explored three essential tools for managing Python packages: `pip` and `virtualenv`. `pip` simplifies package installation, `virtualenv` provides isolation for your Python projects, and `PyPI` serves as a central repository for Python packages. By using these tools effectively, you can manage dependencies and create clean, isolated development environments for your Python projects.

Chapter 30: Context Managers and Magic Methods

In this chapter, we will go through some miscellaneous concepts of python.

Context Manager:

Whenever we write any program, we commonly use computer resources like Files, Network Connections & Database connections. However, such resources are limited and multiple processes need to share it with each other. Whenever our program uses any resource, it must release it, otherwise other processes will keep waiting for the resource and result in timeout. This problem is known as **Resource Leakage**.

Python provides a clean way of releasing resources which is called `Context manager`. A context manager is an object that manages the lifecycle of the resource, ie loading it into our program and releasing it automatically once we are done using it.

Python context managers are commonly used with the `with` statement. The `with` statement ensures that the resource is properly set up before entering the block and properly cleaned up after exiting the block, even if an exception is raised. We will cover some ways of creating our own context managers in this chapter.

Built-in Context Managers:

Python provides several built-in context managers. Let's look at a common one: opening and reading a file.

```
# Using a context manager to open and read a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# File is automatically closed when exiting the block
```

In this example, the `open` function returns a file object that acts as a context manager. The `with` statement ensures that the file is properly closed after reading its content.

Using `contextlib`:

The `contextlib` module provides utilities for creating context managers. You can use the `contextlib.contextmanager` decorator to define a custom context manager using a generator function.

```
from contextlib import contextmanager

@contextmanager
def my_context_manager():
    print("Entering the context")
    # Setup code
    f = open("example.txt", "r")
    yield f
    # Teardown code
    f.close()
    print("Exiting the context")

with my_context_manager() as f:
    # Code inside the context
    print(f.read())
    print("Inside the context")
```

Output:

```
Entering the context
<Content of example.txt>
Inside the context
Exiting the context
```

In this example, the `my_context_manager` function serves as a custom context manager. The `@contextmanager` decorator is used to define the generator. The `yield` statement separates the setup and teardown code, and the code inside the `with` block runs within the managed context.

Creating Custom Context Managers:

To create your own custom context manager, you need to define a class with `__enter__` and `__exit__` methods. Here's an example of a custom context manager for measuring code execution time:

```
import time

class TimerContextManager:
    def __enter__(self):
        self.start_time = time.time()
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.end_time = time.time()
        elapsed_time = self.end_time - self.start_time
        print(f"Execution time: {elapsed_time} seconds")

# Using the custom context manager
with TimerContextManager():
    for _ in range(1000000):
        pass
```

Output:

```
Execution time: 0.03353691101074219 seconds
```

In this example, the `TimerContextManager` class defines the `__enter__` and `__exit__` methods. The `__enter__` method records the start time, and the `__exit__` method calculates and prints the execution time when exiting the block.

Handling Database connection using Context Manager:

Lets understand by an example, how can we manage a database connection using context manager. In below program, we are using a simple database `sqlite3` which comes with Python.

First lets understand an example without context manager and its problem.

```
import sqlite3

# Open connection
connection = sqlite3.connect("context.db")
cur = connection.cursor()

# Create table
cur.execute("CREATE TABLE books(title, topic, price)")
# Insert data
cur.execute("""
    INSERT INTO books VALUES
```

```
        ('Programming in Python', 'Python', 699),
        ('Let us C', 'C', 750)
""")
# Commit
connection.commit()

# Fetch the data
res = cur.execute("SELECT * FROM books")

print(res.fetchall())

# Close the connection manually
connection.close()
```

Output:

```
[('Programming in Python', 'Python', 699), ('Let us C', 'C', 750)]
```

In above example, we opened a database connection and performed some SQL operations. At the end, we closed the connection.

But there is a problem in this code, suppose we get any unexpected error in between, we will never reach to the line where we are closing the connection. We might resolve it using a `try-except-finally` block, but that will need multiple logical changes.

Lets see how we can use context manager to resolve this issue:

```
import sqlite3

with sqlite3.connect("context.db") as connection:
    # Create cursor
    cur = connection.cursor()
    # Create table
    cur.execute("CREATE TABLE books(title, topic, price)")
    # Insert data
    cur.execute("""
        INSERT INTO books VALUES
            ('Programming in Python', 'Python', 699),
            ('Let us C', 'C', 750)
    """)
    # Commit
    connection.commit()
    # Fetch data
    res = cur.execute("SELECT * FROM books")
    print(res.fetchall())
```

Output:

```
[('Programming in Python', 'Python', 699), ('Let us C', 'C', 750)]
```

In above code, we have opened the connection as a context manager and we have just performed the SQL operations without explicitly closing the connection. Context Manager pattern will automatically take care of it even if we don’t handle any exception.

Magic Methods in Python:

Python magic methods, also known as special methods or “dunder methods” (due to their double underscore prefix and suffix), are a fundamental part of Python's object-oriented programming. These methods allow you to define how your custom objects behave in various contexts, such as arithmetic operations, string representations, and more. In this tutorial, we'll explore some common magic methods and how to use them in your Python classes.

What are Magic Methods?

Magic methods are special methods in Python that enable you to define how objects of your class behave in specific situations. They have double underscores at the beginning and end of their names (e.g., `__init__`, `__str__`, `__add__`). When you invoke built-in functions or perform operations on objects, Python automatically calls the relevant magic methods to provide the expected behavior.

Commonly Used Magic Methods:

Python has a vast list of magic methods and we need to know about only few of them. Rest methods are only used in certain situations.

`__init__()` method:

The `__init__` method is used for object initialization. It gets called when you create a new instance of the class.

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(42)
```

`__str__()` method:

The `__str__` method is used to define a string representation of your object when using `str()` or `print()`.

```
class MyClass:
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return f"MyClass instance with value {self.value}"

obj = MyClass(42)
print(obj)
```

Output:

```
MyClass instance with value 42
```

`__repr__` method:

The `__repr__` method is used to define a string representation of your object when using `repr()` or in interactive environments like the Python REPL.

```
class MyClass:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return f"MyClass({self.value})"

obj = MyClass(42)
print(repr(obj))
```

Output:

```
MyClass(42)
```

`__add__()` method:

The `__add__` method allows you to define the behavior of the `+` operator when applied to objects of your class.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x other.x, self.y other.y)
        else:
            raise TypeError("Unsupported operand type for ")

p1 = Point(1, 2)
p2 = Point(3, 4)
result = p1 + p2
print(result.x, result.y)
```

Output:

```
4 6
```

Magic Methods for Comparison

You can define custom behavior for comparison operators using the below magic methods:

Operator	Method
`==`	`__eq__`
`!=`	`__ne__`
`<`	`__lt__`
`>`	`__gt__`
`<=`	`__le__`
`>=`	`__ge__`

Magic Methods for Container Types

For custom container classes like lists or dictionaries, you can define magic methods to control how they behave:

Method	Description
`__len__(self)`	Size of container: `len(list)`
`__getitem__(self, idx)`	Read the object at index `idx`: `list[idx]`
`__setitem__(self, idx, value)`	Write the `value` at index `idx`: `list[idx] = value`

Conclusion

Python context managers simplify resource management by ensuring proper setup and teardown of resources, making your code more readable and robust. Also, python magic methods provide a way to customize the behavior of your custom objects in various contexts, making your classes more powerful and flexible. By implementing these special methods, you can make your code more readable and intuitive, mimicking the behavior of built-in Python types.

Chapter 31: Introduction to PEP8 & Best Practices in Python

PEP 8 Style Guide

Python is known for its readability and simplicity. Whenever we write code in any language, we only write it once but the developer and peers review it multiple times. A code is often visited multiple times to debug issues, performance enhancement or to understand the business logic. Hence it is very important that whatever code we write, be it any language, our code should be readable.

PEP 8, short for Python Enhancement Proposal 8, is a style guide that helps maintain code consistency in the Python community. Following PEP 8 and adopting best practices is essential for writing clean, maintainable, and readable Python code. PEP 8 is a vast documentation, so here we will see some of the most important styles that every developer should follow to maintain a clean code.

1. Indentation and Whitespace

PEP 8 recommends using 4 spaces for indentation and avoiding tabs. Consistency in indentation is crucial for code readability. Also, use blank lines to separate functions, classes, and logical blocks within your code.

```
# Wrong
def foo():
    return True

# Correct
def foo():
  return True
```

2. Maximum Line Length

PEP 8 suggests keeping lines of code within 79 characters. This helps ensure code is readable on various displays without horizontal scrolling. For long lines, you can use Python's implicit line continuation with parentheses, brackets, or backslashes.

3. Import Statements

Import statements should be grouped in the following order:

1. Standard library imports.
2. Related third-party imports.
3. Local application/library specific imports.

Each group should be separated by a blank line. Avoid using wildcard imports ('from module import *') as they can lead to naming conflicts. Below is the example of a good style of import.

```
import sys
import os

import numpy as np
import pandas as pd

from my_module import CustomClass
```

4. Naming Conventions

Follow these common naming conventions:

- Use `snake_case` for functions, variables, and module names.
- Use `CamelCase` for class names.
- Use `UPPERCASE` for constants.

```
# snake_case for functions, variables
def convert_string_to_number(n):
  return int(n) if n.is_digit() else None

# CamelCase
class ErrorHandler:
  def __init__(self, message):
    this.message = message

# UPPERCASE for Constants
PI = 3.1415
```

Choosing descriptive variable and function names makes your code self-documenting and easier to understand.

5. Comments and Docstrings

It is always a good practice to write clear and concise explanation of what your code is doing. The reader will not need to go through line by line and understand the working. Use docstrings to document modules, classes, functions, and methods.

```
# Example with docstring
def is_even_number(n):
  """
  This function checks if the given number is even or not
  """
  # Check if number is valid
  if not isinstance(n, int):
    return False
  return n%2 == 0
```

6. Avoid Extraneous Whitespace

Don't use extraneous whitespace, especially at the end of lines, to keep code clean and easy to read.

7. Operators and Expressions

Use whitespace around operators to improve code readability. For example, write `x = 5` instead of `x=5`. Also, avoid unnecessary parentheses in expressions.

8. Error Handling

Use `try`, `except`, and `finally` blocks for proper error handling. Be specific about the exceptions you catch rather than using a generic `except` clause.

```
# Wrong:
try:
  n = int(input())
  m = int(input())
  print(n/m)
except:
  print("Error occurred")

# Correct
try:
  n = int(input())
  m = int(input())
  print(n/m)
except ZeroDivisionError as ze:
  print("Can't Divide by Zero")
except Exception as e:
```

```
  print("Some unknown error occured", e)

```

9. Use List and Dictionary Comprehensions

Python offers concise ways to create lists and dictionaries using comprehensions. They are more readable than traditional loops and often more efficient.

```
# Below is not a good style
l = []
for i in range(10):
  l.append(i)

# Better way
l = [i for i in range(10)]
```

10. Avoid Using Global Variables

Global variables make code harder to reason about. Instead, use function arguments and return values to pass data between functions. It is always a good practice to take the input in function as argument. It makes your function portable and reusable in multiple places.

11. Consistency of Code

Some organizations have their own style of coding. They follow some self rules and it is very important to observe and follow the existing coding style and conventions. Consistency across the codebase is vital for readability and maintainability.

12. Automate Code Formatting

Tools like `autopep8` and `black` can automatically format your code to adhere to PEP 8 standards. Incorporate these tools into your workflow to ensure consistent code style.

13. Learn from Open Source

An Open source code is the best place to learn new and efficient coding style. Study well-established open-source Python projects to learn from experienced developers. You can gain insights into best practices and coding style by examining their code.

There are many such guidelines but we have covered the most important once in above points. If you want to explore more, feel free to visit the official PEP8 documentation here.

Best Practices of Python

Here are some of the best practices we often use while writing an efficient code in Python. These are based on experiences of developers and can be useful in multiple places.

- Always follow PEP8 for readable code. We have covered it in last section.
- **Using in built functions:** Python is written in C. Hence all of the inbuilt functions internally use C code to perform the operations. C is one of the fastest languages, which makes our Python code much faster if we use in built functions.

```
# Code with poor performance
i = 0
while i < 100000:
  print(i)
  i+=1

# Better Performance when using range() function
for i in range(100000):
  print(i)
```

- **Usage of Third Party Libraries:** Python is Open source. Most of the third party libraries are also Open source in Python. The Open source code is often written by brilliant and experienced developers, and it is better in terms of performance and memory usage. These codes are also maintained and regularly updated for issues. Hence it is always a good practice to reuse it rather than creating your own modules and repeating the cycle. We can always install any open source module using `pip.`

```
pip install <module-name>
```

- **Generators:** Generators are memory efficient and whenever we don't want to load the entire list of item in memory, we should always go for generators.
- **Time Complexity:** It is a good practice to understand the time complexity of our code and try to optimise it wherever possible.
- **Caching:** Some functions might be called multiple times in a code. There is a large possibility in real life projects that we call a function with same parameters over and over and unnecessarily compute the output again and again. Wherever possible, we should try to cache the output. Below is a in-built library in Python to make your functions cached:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def get_name_from_user_id(user_id):
    result = database.get_data(user_id)
    return result.name
```

- **Avoid Global Variables:** We should always create functions that are independent of any outer object. Each function should be portable and reusable from anywhere. Hence we should always avoid global variables.
- **List Comprehensions:** A list comprehension is faster than traditional for loops while creating a list. Always try to use list comprehension to avoid appending the data one by one.
- **Use of Sets and Dictionaries:** Set and Dictionary data structures provide fast constant time lookups. Wherever needed, we should use these to avoid long lookup times.
- **Immutable Objects:** Immutable objects are created once in the memory and they don't change. They are easy to manage and hence they are performance wise efficient. Whenever we are sure that some objects are not gonna change in their lifecycle, we should go for Immutable objects like tuples, Strings etc.
- **Local Imports:** Always import the useful code instead of importing the entire module. It reduces the import overhead.

There can be so many optimizations we can apply on our code and make our programs faster. But such things are better learned from experience and practice. In this chapter we have seen some of the PEP8 styling rules to make our code maintainable and clean. We have also gone through some best practices while coding in Python.