

Contents

Fundamentals of Python	4
Basics of programming	4
Types of programming languages	4
Python Programming Language	4
Executing python programs	4
Python Features	6
Data Types & Operators in Python	7
Variables	7
Input and Output in Python	8
Working with numbers:	10
Example Programs	10
Working with strings	12
Typecasting	13
Additional Concepts	15
Operators	15
Python Operators	15
Misc Concepts	19
Short Circuit Evaluation	19
Lists in Python	20
List Slicing	21
Dictionaries in Python	22
Sets in Python	23
Control Flow Statements in Python	25
1. if Statement	25
2. for Loop	25
Using the range() Function	25
Decremental Loop Example	26
3. while Loop	26
Infinite Loops	26
4. break and continue	27
Functions in Python	28
Defining a Function	28
Calling a Function	28
Function Parameters and Return Values	28
Default Parameters	28

Keyword Arguments	29
Variable Length Arguments	29
Combining Different Types of Arguments	29
Structure of a Python Script	30
Scope of Variables	31
Local Scope	31
Global Scope	31
The <code>global</code> Keyword	31
The <code>nonlocal</code> Keyword	32
Command Line Arguments	32
Useful Built-in Functions	33
<code>zip</code>	33
<code>enumerate</code>	33
<code>sorted</code>	34
<code>filter</code>	34
<code>map</code>	34
Modules & Packages in Python	35
Modules	35
Packages	35
Aliasing Modules & Imports	35
Exploring Modules with <code>help()</code> and <code>dir()</code>	36
Exploring Built-in Modules	36
The <code>string</code> Built-in Module	37
Exploring & Using the <code>math</code> Module	37
Error Handling & Debugging in Python	39
Error Handling	39
Raising Exceptions	39
Debugging	39
Using Print Statements	39
The <code>pdb</code> Debugger	39
Common Debugging Tools	40
File I/O in Python	41
Opening a File	41
Reading from a File	41
Writing to a File	41
Closing a File	42
Example: Copying a File	42
Summary	42

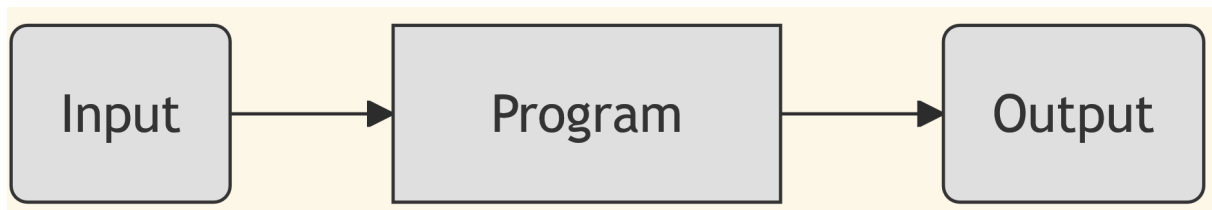
Object Oriented Programming (OOP) Concepts in Python	43
Key Concepts	43
1. Class	43
2. Object	43
3. Inheritance	43
4. Encapsulation	44
5. Polymorphism	44
Summary	44
Advanced Python Concepts	46
Generators	46
Iterators	46
Decorators	47
Dataclasses	47
List Comprehension	48
Dictionary Comprehension	48
__slots__	49

Fundamentals of Python

Basics of programming

How do you approach a programming problem?

Think of a program as a black box which transforms an input to a specified output.



Researchers have found that more than 40% of the time is invested in coding in a overall project schedule. Finding logical solutions to problems in a language a computer understands means programming.

Types of programming languages

1. Compiled Languages: C, C++ etc..
Source code --> compile --> Binary Code --> execute
2. Interpreted languages: Python, Javascript, Java etc..
Source Code --> interpret(execute) Source Code --> Byte Code --> interpret

Python Programming Language

- Python is an **interpreted** language like Java
- It needs an interpreter to run
- It was invented by **Guido Van Rossum** in **1991**
- Current Version of python at the time of writing is **3.13.3**
- You need to install python Interpreter before you can execute python programs
- Interpretation means executing commands line by line
- It is a **platform-independent** language
- It is a **WORA - Write Once Run Anywhere** kind of language
- The python interpreter is tied to the OS architecture
- There are different kinds of python interpreters - **Jython, Cython, Pypi**

Executing python programs

To execute python programs, we need python interpreter, it can be installed from python.org.

- Interactive mode of execution

```
~$ python3
Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> 2 + 4
6
>>> 35 ** 2
1225
>>> quit()
~$
```

- Script Mode of execution

```
23 # Script to calculate Simple Interest
24
25 import sys
26
27 def main(args):
28     principal = 100000
29     rate = 12
30     years = 3
31     si = (principal * rate * years) / 100
32     print("Simple Interest is: " + str(si))
33     return 0
34
35
36 if __name__ == '__main__':
37     sys.exit(main(sys.argv[1:]))
38
```

```
C:\WINDOWS\SYSTEM32\cmd.exe
Simple Interest is: 36000.0
-----
(program exited with code: 0)
Press any key to continue . . .
```

- Notebook mode of execution

Chapter 1

Summary

- Variables and Naming conventions
- Input and output
- Operators

Creating a variable

```
x = 10
print(x) #This is an example of comment
```

[1] Python

... 10

Python Features

1. Portability
2. Object Oriented & Functional
3. Scalable
4. Interfacing with other languages & tools
5. Built on C API
6. Dynamic Programming
7. Ease of use & debugging

Data Types & Operators in Python

Python has 5 basic data types:

Data Type	Description	Example
number	Any integer or float (+ve or -ve)	12,-123,34.456, -56.89
string	anything enclosed in single or double quotes	'abcd', "HP", 'computer'
list	dynamic array	[1,2,3,45,"abd",[4,5,6]]
Tuple	static array	(1,2,3), ("ab","cd","ef")
Dictionary	Key value pairs	{'eno':101,'ename':'john doe','esalary':20000}

Variables

- The smallest element that can hold data.
- Depending on the value stored , the type of variable changes accordingly.
- It represents a memory location.
- It can be any word other than the keywords.
- = operator is used to assign value to a variable. To create a variable

Syntax:

```
var_name = value
```

Rules for creating variables:

1. no declaration (only assignment)
2. case sensitive
3. Variables created in a script are global
4. A variable is represented as id in the memory, to know we can use `id()` method

```
# Creating a variable
ename = "John Doe"

# To know the type of the variable
type(ename)
```

```
# To check the id of the variable
id(ename)

# Assigning one variable to another
emp_name = ename

# Check the id of new variable
id(emp_name)

# To delete the variable
del(ename)
```

Input and Output in Python

Python provides simple functions for input and output (I/O) operations.

Output with `print()` The `print()` function displays data to the standard output (usually the screen).

Example:

```
print("Hello, World!")
name = "Alice"
print("Hello,", name)
```

You can print multiple values, use separators, and change the end character:

```
print("A", "B", "C", sep="-")    # Output: A-B-C
print("Done", end="!")          # Output: Done!
```

Printing with Format Specifiers and f-Strings Python offers multiple ways to format strings when printing variables and values.

1. Using Format Specifiers with `format()`

The `str.format()` method allows you to insert values into a string using curly braces `{}` as placeholders.

Example:


```
name = "Alice"
age = 25
print("Name: {}, Age: {}".format(name, age))
```

You can also use positional or named placeholders:

```
print("Name: {0}, Age: {1}".format(name, age))
print("Name: {n}, Age: {a}".format(n=name, a=age))
```

You can specify formatting options, such as number of decimal places:

```
pi = 3.14159
print("Value of pi: {:.2f}".format(pi))  # Output: Value of pi: 3.14
```

2. Using f-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings provide a concise way to embed expressions inside string literals, using {}.

Example:

```
name = "Alice"
age = 25
print(f"Name: {name}, Age: {age}")
```

You can include expressions and formatting directly:

```
pi = 3.14159
print(f"Value of pi: {pi:.2f}")  # Output: Value of pi: 3.14
print(f"Next year, {name} will be {age + 1} years old.")
```

Input with input() The `input()` function reads a line of text from the user as a string.

Example:

```
name = input("Enter your name: ")
print("Hello,", name)
```

To read numbers, convert the input string:

```
age = int(input("Enter your age: "))
price = float(input("Enter price: "))
```

Working with numbers:

```
>>> x = 123
>>> type(x)
<class 'int'>
>>> id(x)
140293271359536
>>> z = float(x)
>>> x
123
>>> z
123.0
>>> y = 5
>>> x + y
128
>>> x - y
118
>>> x * y
615
>>> x / y
24.6
>>> x // y Integer Division
24
>>> x % y
3
>>> x ** y x to the power y
28153056843
>>>
```

Example Programs

```
principal = 100000
rate = 12
years = 3
si = (principal * rate * years) / 100
print("Simple Interest is: " + str(si))
```

To Calculate the Simple Interest Output

```

23 # Script to calculate Simple Interest
24
25 import sys
26
27 def main(args):
28     principal = 100000
29     rate = 12
30     years = 3
31     si = (principal * rate * years) / 100
32     print("Simple Interest is: " + str(si))
33     return 0
34
35
36 if __name__ == '__main__':
37     sys.exit(main(sys.argv[1:]))
38

```

```

C:\WINDOWS\SYSTEM32\cmd.exe
Simple Interest is: 36000.0

```

```

-----
(program exited with code: 0)
Press any key to continue . . .

```

```

basic = 10000
hra = .4 * basic
da = .2 * basic
gross = (basic + hra + da)
tax = .1 * basic
net = gross - tax

print("Net Salary of an employee is: " + str(net))

```

Computing net salary of an employee Output

```

22 # Script to calculate net salary of an employee
23
24
25 import sys
26
27 def main(args):
28     basic = 10000
29     hra = .4 * basic
30     da = .2 * basic
31     gross = (basic + hra + da)
32     tax = .1 * basic
33     net = gross - tax
34
35     print("Net Salary of an employee is: " + str(net))
36     return 0
37
38

```

```

C:\WINDOWS\SYSTEM32\cmd.exe
Net Salary of an employee is: 15000.0

```

```

-----
(program exited with code: 0)
Press any key to continue . . .

```

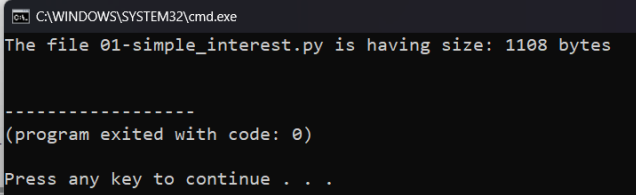
```

fname = "01-simple_interest.py"
fsize = os.path.getsize(fname)
print("The file " + str(fname) + " is having size: " + str(fsize) + "
    bytes")

```

Displaying the size of a file Output

```
22 # Script to display size of a file
23
24
25 import sys
26 import os
27
28 def main(args):
29     fname = "01-simple_interest.py"
30     fsize = os.path.getsize(fname)
31     print("The file " + str(fname) + " is having size: " +
32         str(fsize) + " bytes")
33     return 0
```



Working with strings

Anything which is enclosed in single or double quotes is a string. Multi-line strings are created using three single quotes or 3 double quotes at the beginning & at the end. A string is unlimited & immutable (read-only).

Example: 'abcd', "HP"

```
>>> x = 'computer'
>>> type(x)
<class 'str'>
>>> id(x)
139905856163824
>>> len(x)
8
>>> max(x)
'u'
>>> min(x)
'c'
>>> x[0]
'c'
>>> x[3]
'p'
>>> x[-1]
'r'
>>> x[-4]
'u'
>>>
```

String Slicing String slicing in Python allows you to extract a portion (substring) of a string using the syntax `string[start:stop:step]`. The start index is inclusive, stop is exclusive, and step

determines the stride. If omitted, defaults are `start=0`, `stop=len(string)`, and `step=1`.

Example:

```
text = "Hello, World!"
print(text[0:5])      # Output: Hello
print(text[7:])       # Output: World!
print(text[::-1])     # Output: !dlroW ,olleH
```

```
>>> x = 'computer'
>>> x[2:]
'mputer'
>>> x[:2]
'co'
>>> x[-3:]
'ter'
>>> x[:-3]
'compu'
>>> x[2:5]
'mpu'
>>> x[2:3]
'm'
>>> x[2:-3]
'mpu'
>>> x[:]
'computer'
>>> x[: :2]
'cmue'
>>> x[: :-1]
'retupmoc'
>>> x[: :-2]
'rtpo'
>>> x[1: :2]
'optr'
>>> █
```

Typecasting

Typecasting is the process of converting a value from one data type to another. This is often necessary when you want to perform operations between different types, or when a function expects a specific

type as input.

There are two main types of typecasting:

Implicit Typecasting (Automatic Conversion):

The programming language automatically converts one data type to another. Example: In Python, adding an integer and a float will automatically convert the integer to a float.

Explicit Typecasting (Manual Conversion):

The programmer manually converts one data type to another using casting functions or syntax. Example: `float(5)` converts the integer 5 to a float 5.0.

Why is typecasting important?

Prevents type errors in your code. Ensures data is in the correct format for operations or function calls. Helps with interoperability between different parts of a program. Gotcha: Be careful when typecasting, as converting between incompatible types (e.g., a string that doesn't represent a number to an integer) can cause runtime errors. Always validate or handle exceptions when typecasting user input.

```
>>> a = 123
>>> type(a)
<class 'int'>
>>> b = '456'
>>> type(b)
<class 'str'>
>>> c = a + b
      File "<stdin>", line 1
        = a + b
IndentationError: unexpected indent
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> c = a + int(b)
>>> d = str(a) + b
>>> c
579
>>> d
'123456'
>>> █
```

Additional Concepts

```
>>> a = 1
>>> chr(a)
'\x01'
>>> b = '1'
>>> ord(b)
49
>>> ord('a')
97
>>> 'a' + 'b'
'ab'
>>> 'a' * 12
'aaaaaaaaaaaa'
>>> '-'*12
'-----'
>>>
```

byte value of an integer

ASCII Value of a character

Adding 2 characters concatenates them

Multiply a str by integer repeats the character

```
>>> chr(65)
'A'
>>> chr(49)
'1'
>>>
```

Operators

Python Operators

Operators are special symbols or keywords in Python that are used to perform operations on variables and values. Python supports several types of operators:

Arithmetic Operators

Used for mathematical operations:

Operator	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction	5 - 2	3
*	Multiplication	3 * 4	12
/	Division	10 / 2	5.0
//	Floor Division	7 // 2	3
%	Modulus	7 % 2	1
**	Exponentiation	2 ** 3	8

Comparison Operators

Used to compare values, returning True or False:

Operator	Description	Example	Result
==	Equal to	3 == 3	True
!=	Not equal to	3 != 4	True
>	Greater than	5 > 2	True
<	Less than	2 < 5	True
>=	Greater or equal	5 >= 5	True
<=	Less or equal	2 <= 3	True

Assignment Operators

Used to assign values to variables:

Operator	Example	Equivalent to
=	a = 5	
+=	a += 2	a = a + 2
-=	a -= 2	a = a - 2
*=	a *= 3	a = a * 3
/=	a /= 2	a = a / 2
//=	a //= 2	a = a // 2
%=	a %= 2	a = a % 2
**=	a **= 2	a = a ** 2

Logical Operators

Used to combine conditional statements:

Operator	Description	Example	Result
and	Logical AND	True and False	False
or	Logical OR	True or False	True

Operator	Description	Example	Result
not	Logical NOT	not True	False

Membership Operators

Test for membership in a sequence:

Operator	Description	Example	Result
in	Value present	'a' in 'cat'	True
not in	Value not present	'x' not in 'cat'	True

Identity Operators

Compare memory locations of two objects:

Operator	Description	Example	Result
is	Same object	a is b	True/False
is not	Not same object	a is not b	True/False

Bitwise Operators

Operate on binary representations:

Operator	Description	Example	Result
&	AND	5 & 3	1
	OR	5 3	7
^	XOR	5 ^ 3	6
~	NOT	~5	-6
<<	Left Shift	5 << 1	10
>>	Right Shift	5 >> 1	2

Example:

```
a = 10
b = 3
print(a + b)      # 13
print(a > b)      # True
print(a & b)       # 2
print(a is b)     # False
```

```
>>> a = 10
>>> b = 3
>>> print( a + b)
13
>>> print(a > b)
True
>>> print(a & b)
2
>>> print( a is b)
False
>>> █
```

Operators are fundamental to writing expressions and controlling the flow of your Python programs.

Misc Concepts

```
>>> 1 and 0
0
>>> 0 and 2
0
>>> 2 and 6
6
>>> 5 and 3
3
>>> 1 or 0
1
>>> 0 or 2
2
>>> 5 or 3
5
>>> 9 or 8
9
>>> █
```

Short Circuit Evaluation

Short circuit evaluation refers to the way logical operators `and` and `or` work in Python. When evaluating expressions, Python stops as soon as the result is determined:

- For `and`, if the first operand is `False`, the whole expression is `False` and the second operand is not evaluated.
- For `or`, if the first operand is `True`, the whole expression is `True` and the second operand is not evaluated.

Example:

```
def check():
    print("Function called")
    return True

# 'check()' is not called because the first condition is False
if False and check():
    print("Won't print")
```

```
# 'check()' is called because the first condition is True
if True or check():
    print("Will print")
```

Guard Pattern The guard pattern uses short circuit evaluation to prevent errors by ensuring that certain conditions are met before executing code that could fail. It's commonly used to check for None or validate input before accessing attributes or methods.

Example:

```
user = None

# Guard pattern: Only access 'name' if 'user' is not None
if user is not None and user.name == "Alice":
    print("Hello, Alice!")
```

Here, `user.name` is only accessed if `user` is not `None`, preventing an `AttributeError`.

Short circuiting and guard patterns help write safer and more efficient code.

```
>>> x = 6
>>> y = 2
>>> x / y
3.0
>>> y = 0
>>> x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> y != 0 and x / y
False
>>> y = 2
>>> y != 0 and x / y
3.0
>>> █
```

Lists in Python

A **list** is a built-in data type in Python used to store an ordered collection of items, which can be of different types (integers, strings, other lists, etc.). Lists are mutable, meaning you can change their contents after creation.

Creating a List:

```
numbers = [1, 2, 3, 4, 5]
mixed = [1, "apple", 3.14, [2, 4]]
```

Accessing Elements:

- Use zero-based indexing: `numbers[0]` gives 1
- Negative indices count from the end: `numbers[-1]` gives 5

Modifying Lists:

```
numbers[2] = 10      # Change value at index 2
numbers.append(6)     # Add an item to the end
numbers.insert(1, 7)  # Insert 7 at index 1
del numbers[0]        # Remove item at index 0
```

Common List Methods:

Method	Description	Example
<code>append(x)</code>	Add item to end	<code>numbers.append(10)</code>
<code>insert(i, x)</code>	Insert item at position <i>i</i>	<code>numbers.insert(1, 20)</code>
<code>remove(x)</code>	Remove first occurrence of <i>x</i>	<code>numbers.remove(3)</code>
<code>pop([i])</code>	Remove and return item at index <i>i</i>	<code>numbers.pop()</code>
<code>sort()</code>	Sort the list in place	<code>numbers.sort()</code>
<code>reverse()</code>	Reverse the list in place	<code>numbers.reverse()</code>

Iterating Over a List:

```
for item in numbers:
    print(item)
```

Lists are one of the most versatile and commonly used data structures in Python.

List Slicing

List slicing in Python allows you to extract a portion (sublist) of a list using the syntax `list[start:stop:step]`. The `start` index is inclusive, `stop` is exclusive, and `step` determines the stride. If omitted, defaults are `start=0`, `stop=len(list)`, and `step=1`.

Example:

```
numbers = [10, 20, 30, 40, 50, 60]

print(numbers[1:4])    # [20, 30, 40]
print(numbers[:3])     # [10, 20, 30]
print(numbers[::2])    # [10, 30, 50]
print(numbers[::-1])   # [60, 50, 40, 30, 20, 10]
```

Slicing is a powerful way to access and manipulate sublists without modifying the original list.

Dictionaries in Python

A **dictionary** is a built-in data type in Python that stores data as key-value pairs. Dictionaries are unordered (prior to Python 3.7), mutable, and indexed by keys, which must be immutable types (like strings, numbers, or tuples).

Creating a Dictionary:

```
student = {
    "name": "Alice",
    "age": 20,
    "marks": [85, 90, 92]
}
```

Accessing Values:

- Use the key inside square brackets: `student["name"]` gives "Alice"
- Use `.get()` to avoid errors if the key is missing: `student.get("grade")` returns None

Modifying Dictionaries:

```
student["age"] = 21          # Update value
student["grade"] = "A"       # Add new key-value pair
del student["marks"]         # Remove a key-value pair
```

Common Dictionary Methods:

Method	Description	Example
<code>keys()</code>	Returns all keys	<code>student.keys()</code>
<code>values()</code>	Returns all values	<code>student.values()</code>
<code>items()</code>	Returns all key-value pairs as tuples	<code>student.items()</code>
<code>get(key)</code>	Returns value for key or None	<code>student.get("name")</code>
<code>update(dict2)</code>	Updates dictionary with another dictionary	<code>student.update({"age": 22})</code>
<code>pop(key)</code>	Removes key and returns its value	<code>student.pop("grade")</code>

Iterating Over a Dictionary:

```
for key, value in student.items():  
    print(key, value)
```

Dictionaries are ideal for representing structured data and for fast lookups by key.

Sets in Python

A **set** is a built-in data type in Python used to store an unordered collection of unique items. Sets are mutable, but their elements must be immutable types (like numbers, strings, or tuples).

Creating a Set:

```
fruits = {"apple", "banana", "cherry"}  
empty_set = set() # Note: {} creates an empty dictionary, not a set
```

Key Properties:

- No duplicate elements: Each item appears only once.
- Unordered: No indexing or slicing.
- Mutable: You can add or remove elements.

Common Set Methods:

Method	Description	Example
<code>add(x)</code>	Add element x to the set	<code>fruits.add("orange")</code>
<code>remove(x)</code>	Remove element x (error if absent)	<code>fruits.remove("banana")</code>
<code>discard(x)</code>	Remove element x (no error)	<code>fruits.discard("pear")</code>
<code>pop()</code>	Remove and return an arbitrary item	<code>fruits.pop()</code>
<code>clear()</code>	Remove all elements	<code>fruits.clear()</code>
<code>copy()</code>	Return a shallow copy	<code>fruits.copy()</code>

Set Operations: Sets support mathematical operations like union, intersection, difference, and symmetric difference.

Operation	Syntax	Example Result
Union	<code>set1 set2</code>	Items in both sets
Intersection	<code>set1 & set2</code>	Items in both sets
Difference	<code>set1 - set2</code>	Items in set1 not in set2
Symmetric Diff.	<code>set1 ^ set2</code>	Items in either, not both

Example:

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b) # {1, 2, 3, 4, 5}
print(a & b) # {3}
print(a - b) # {1, 2}
print(a ^ b) # {1, 2, 4, 5}
```

Sets are useful for membership testing, removing duplicates, and performing set-theoretic operations.

Control Flow Statements in Python

Control flow statements allow you to control the execution order of statements in your code. The main types are: `if`, `for`, and `while`.

1. `if` Statement

Used for conditional execution.

```
x = 10
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is 5")
else:
    print("x is less than 5")
```

2. `for` Loop

Used to iterate over a sequence.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Using the `range()` Function

The `range()` function generates a sequence of numbers, commonly used with `for` loops.

Syntax:

`range(stop)`

`range(start, stop)`

`range(start, stop, step)`

- `start`: Starting value (default is 0)
- `stop`: End value (not included)
- `step`: Increment (default is 1)

Example:

```
for i in range(3):  
    print(i) # Outputs 0, 1, 2  
  
for i in range(1, 6, 2):  
    print(i) # Outputs 1, 3, 5
```

Decremental Loop Example

You can use a for loop with a negative step in range() to count down.

```
for i in range(5, 0, -1):  
    print(i) # Outputs 5, 4, 3, 2, 1
```

3. while Loop

Repeats as long as a condition is true.

```
count = 0  
while count < 3:  
    print("Count:", count)  
    count += 1
```

Infinite Loops

An infinite loop is a loop that never ends because its condition always evaluates to True. Infinite loops are useful in situations where you want your program to keep running until it is explicitly stopped, such as in servers or waiting for user input.

Example:

```
while True:  
    print("This will run forever unless stopped manually.")
```

Be careful with infinite loops, as they can cause your program to become unresponsive if not handled properly. Always ensure there is a way to break out of the loop when needed.

4. break and continue

- break exits the loop.
- continue skips to the next iteration.

```
for i in range(5):  
    if i == 3:  
        break  
    print(i)
```

```
for i in range(5):  
    if i == 2:  
        continue  
    print(i)
```

Note: Though for loop and while loop can be used to handle looping, the rule of thumb is **When you know the number of steps (iterations) to take, use for loop otherwise (When you dont know the number of steps or you know the exit condition) use while loop**

Functions in Python

Functions are reusable blocks of code that perform a specific task. They help organize code, avoid repetition, and improve readability.

Defining a Function

Use the `def` keyword to define a function:

```
def greet(name):  
    print(f"Hello, {name}!")
```

Calling a Function

Invoke a function by using its name and passing required arguments:

```
greet("Alice")
```

Function Parameters and Return Values

Functions can accept parameters and return values:

```
def add(a, b):  
    return a + b  
  
result = add(3, 5)  # result is 8
```

Default Parameters

You can provide default values for parameters:

```
def greet(name="World"):  
    print(f"Hello, {name}!")
```

Keyword Arguments

Arguments can be passed by name:

```
greet(name="Bob")
```

Variable Length Arguments

Functions can accept a variable number of arguments using `*args` for positional arguments and `**kwargs` for keyword arguments:

```
def print_numbers(*args):
    for number in args:
        print(number)

print_numbers(1, 2, 3) # prints 1, 2, 3

def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)
```

Combining Different Types of Arguments

You can combine fixed (positional), default, variable-length positional (`*args`), and variable-length keyword (`**kwargs`) arguments in a single function. The order should be:

1. Positional arguments
2. Default arguments
3. `*args`
4. `**kwargs`

```
def example(a, b=2, *args, **kwargs):
    print(f"a: {a}")
    print(f"b: {b}")
    print(f"args: {args}")
    print(f"kwargs: {kwargs}")
```

```
example(1, 3, 4, 5, x=10, y=20)
```

Output:

```
a: 1
b: 3
args: (4, 5)
kwargs: {'x': 10, 'y': 20}
```

This allows your function to handle a flexible number and type of arguments.

Structure of a Python Script

A typical Python script consists of the following components:

1. **Shebang Line (optional):** Indicates the interpreter to use when running the script from the command line.

```
#!/usr/bin/env python3
```

2. **Module Imports:** Import necessary modules or packages at the top of the script.

```
import sys
import math
```

3. **Global Variables and Constants:** Define any constants or global variables.

```
PI = 3.14159
```

4. **Function and Class Definitions:** Define reusable functions and classes.

```
def calculate_area(radius):
    return PI * radius ** 2
```

5. **Main Execution Block:** Use the `if __name__ == "__main__":` construct to specify code that should run when the script is executed directly.

```
if __name__ == "__main__":
    print(calculate_area(5))
```

This structure helps organize code, improves readability, and allows code reuse.

Scope of Variables

The scope of a variable determines where it can be accessed within your code.

Local Scope

Variables defined inside a function are local to that function and cannot be accessed outside:

```
def my_function():
    x = 10 # local variable
    print(x)

my_function()
# print(x) # This would cause an error: NameError: name 'x' is not defined
```

Global Scope

Variables defined outside any function are global and can be accessed throughout the script:

```
y = 20 # global variable

def show_y():
    print(y)

show_y() # prints 20
```

The global Keyword

To modify a global variable inside a function, use the global keyword:

```
count = 0

def increment():
    global count
    count += 1

increment()
print(count) # prints 1
```

The nonlocal Keyword

The `nonlocal` keyword allows you to modify a variable in an enclosing (but non-global) scope, such as inside nested functions:

```
def outer():
    x = 5
    def inner():
        nonlocal x
        x = 10
    inner()
    print(x)  # prints 10

outer()
```

Understanding variable scope helps prevent bugs and makes your code easier to maintain.

Command Line Arguments

Python scripts can accept input from the command line using the `sys.argv` list from the `sys` module. This allows you to pass arguments to your script when you run it.

- `sys.argv` is a list where:
 - The first element (`sys.argv[0]`) is the script name.
 - The following elements are the arguments passed to the script.

Example:

Suppose you have a script called `example.py`:

```
import sys

print("Script name:", sys.argv[0])
print("Arguments:", sys.argv[1:])
```

If you run the script from the command line:

```
python example.py hello world
```

Output:

Script name: `example.py`

Arguments: `['hello', 'world']`

You can use these arguments to control your script's behavior, such as processing files or setting options. For more advanced argument parsing, consider using the `argparse` module.

Useful Built-in Functions

Python provides several built-in functions that are commonly used when working with sequences and iterables:

zip

The `zip` function combines two or more iterables (like lists or tuples) element-wise into tuples:

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]
for name, score in zip(names, scores):
    print(f"{name}: {score}")
```

Output:

```
Alice: 85
Bob: 92
Charlie: 78
```

enumerate

The `enumerate` function adds a counter to an iterable, returning pairs of index and value:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

sorted

The `sorted` function returns a new sorted list from the items in any iterable:

```
numbers = [3, 1, 4, 1, 5]
print(sorted(numbers)) # [1, 1, 3, 4, 5]
```

You can also sort by a custom key:

```
words = ["banana", "apple", "cherry"]
print(sorted(words, key=len)) # ['apple', 'banana', 'cherry']
```

filter

The `filter` function constructs an iterator from elements of an iterable for which a function returns `True`:

```
numbers = [1, 2, 3, 4, 5]
even = filter(lambda x: x % 2 == 0, numbers)
print(list(even)) # [2, 4]
```

map

The `map` function applies a function to every item of an iterable and returns an iterator:

```
numbers = [1, 2, 3, 4]
squares = map(lambda x: x ** 2, numbers)
print(list(squares)) # [1, 4, 9, 16]
```

These functions are powerful tools for processing and transforming data in Python.

Modules & Packages in Python

Modules

A **module** is a single Python file (.py) containing definitions, functions, classes, or runnable code. Modules help organize code into reusable components.

Example:

```
# file: mymodule.py
def greet(name):
    print(f"Hello, {name}!")
```

Importing a module:

```
import mymodule
mymodule.greet("Alice")
```

Packages

A **package** is a directory containing multiple modules and a special `__init__.py` file (can be empty). Packages allow hierarchical structuring of modules.

Example structure:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

Importing from a package:

```
from mypackage import module1
module1.some_function()
```

Aliasing Modules & Imports

You can assign an alias to a module or object when importing, making it easier to reference in your code.

Aliasing a module:

```
import mymodule as mm
mm.greet("Bob")
```

Aliasing an imported object:

```
from mymodule import greet as hello
hello("Carol")
```

Aliasing is useful for shortening long module names or avoiding naming conflicts.

Exploring Modules with `help()` and `dir()`

Python provides built-in functions to explore modules and their contents:

- **`help()`**: Displays documentation for modules, functions, classes, or objects.
- **`dir()`**: Lists the names defined in a module or object (such as functions, classes, and variables).

Example usage:

```
import mymodule

help(mymodule)          # Shows documentation for mymodule
print(dir(mymodule))    # Lists all attributes and functions in mymodule
```

These tools are useful for discovering available functionality and understanding how to use modules.

Exploring Built-in Modules

Python comes with a rich set of built-in modules that provide standard functionality. You can explore available built-in modules using the `sys` and `pkgutil` modules.

Listing all built-in modules:

```
import sys

print(sys.builtin_module_names)
```

Finding all available modules (including installed ones):

```
import pkgutil

for module_info in pkgutil.iter_modules():
    print(module_info.name)
```

You can also use `help('modules')` in the Python interactive shell to see a list of all available modules.

These techniques help you discover modules that are ready to use without additional installation.

The string Built-in Module

Python's built-in `string` module provides useful constants and functions for string manipulation and formatting.

Common features:

- Predefined character sets like `ascii_letters`, `digits`, and `punctuation`.
- Utility functions for string formatting and templates.

Example usage:

```
import string

print(string.ascii_lowercase) # 'abcdefghijklmnopqrstuvwxyz'
print(string.digits)         # '0123456789'
```

The `string` module is helpful for tasks like validation, generating random strings, or custom formatting.

Exploring & Using the math Module

The built-in `math` module provides mathematical functions and constants, such as trigonometric functions, logarithms, and the value of `pi`.

Common features:

- Mathematical constants: `math.pi`, `math.e`
- Functions: `math.sqrt()`, `math.sin()`, `math.log()`, `math.factorial()`, etc.

Example usage:

```
import math

print(math.pi)           # 3.141592653589793
print(math.sqrt(16))     # 4.0
print(math.sin(math.pi)) # 0.0
print(math.log(100, 10)) # 2.0
```

Use `dir(math)` or `help(math)` to explore all available functions and constants in the module.

Error Handling & Debugging in Python

Error Handling

Python uses `try`, `except`, `else`, and `finally` blocks to handle exceptions (errors):

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("No errors occurred.")
finally:
    print("This always runs.")
```

- **try:** Code that might throw an error.
- **except:** Handles specific exceptions.
- **else:** Runs if no exceptions occur.
- **finally:** Always runs, for cleanup.

Raising Exceptions

You can raise exceptions using `raise`:

```
if value < 0:
    raise ValueError("Value must be non-negative")
```

Debugging

Using Print Statements

Insert `print()` statements to check variable values and program flow.

The `pdb` Debugger

Python includes a built-in debugger called `pdb`:

```
import pdb; pdb.set_trace()
```

This starts an interactive debugging session.

Common Debugging Tools

- **IDEs:** Most IDEs (like VS Code, PyCharm) have integrated debuggers.
- **Logging:** Use the logging module for more advanced output than `print()`.

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug("This is a debug message")
```


File I/O in Python

File I/O (Input/Output) allows you to read from and write to files on your system.

Opening a File

Use the `open()` function:

```
file = open('example.txt', 'r') # 'r' for read mode
```

Common modes:

- `'r'`: Read (default)
- `'w'`: Write (creates/truncates file)
- `'a'`: Append
- `'b'`: Binary mode (e.g., `'rb'`)

Reading from a File

```
with open('example.txt', 'r') as file:  
    content = file.read()  
    print(content)
```

Other methods:

- `readline()`: Reads one line
- `readlines()`: Reads all lines into a list

Writing to a File

```
with open('example.txt', 'w') as file:  
    file.write('Hello, world!\n')
```

Use `'a'` mode to append instead of overwrite.

Closing a File

Files should be closed after use. Using `with` handles this automatically.

Example: Copying a File

```
with open('source.txt', 'r') as src, open('dest.txt', 'w') as dst:
    for line in src:
        dst.write(line)
```

Summary

- Use `open()` to access files.
- Always close files or use `with` for automatic handling.
- Choose the correct mode for your operation.
- Read and write using provided methods.

Object Oriented Programming (OOP) Concepts in Python

Object Oriented Programming (OOP) is a programming paradigm that organizes code using objects and classes. Python supports OOP, making it easier to structure complex programs.

Key Concepts

1. Class

A class is a blueprint for creating objects. It defines attributes and methods.

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print(f"{self.name} says woof!")
```

2. Object

An object is an instance of a class.

```
my_dog = Dog("Buddy")
my_dog.bark() # Output: Buddy says woof!
```

3. Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

```
class Animal:
    def eat(self):
        print("Eating...")

class Cat(Animal):
    def meow(self):
        print("Meow!")
```

4. Encapsulation

Encapsulation hides the internal state of an object and requires all interaction to be performed through methods.

```
class Counter:
    def __init__(self):
        self.__count = 0 # Private attribute

    def increment(self):
        self.__count += 1

    def get_count(self):
        return self.__count
```

5. Polymorphism

Polymorphism allows different classes to be treated as instances of the same class through a common interface.

```
class Bird:
    def speak(self):
        print("Chirp!")

class Parrot(Bird):
    def speak(self):
        print("Squawk!")

def animal_sound(animal):
    animal.speak()

animal_sound(Bird()) # Output: Chirp!
animal_sound(Parrot()) # Output: Squawk!
```

Summary

- **Class:** Blueprint for objects.
- **Object:** Instance of a class.
- **Inheritance:** Reuse code across classes.

- **Encapsulation:** Hide internal details.
- **Polymorphism:** Use a unified interface for different types.

OOP helps in organizing code, making it reusable and easier to maintain.

Advanced Python Concepts

Generators

Generators are special functions that yield values one at a time using the `yield` keyword. They allow you to iterate over large data sets without loading everything into memory.

```
def countdown(n):  
    while n > 0:  
        yield n  
        n -= 1  
  
for num in countdown(5):  
    print(num)
```

Advantages:

- Memory efficient
 - Lazy evaluation (values produced on demand)
-

Iterators

An iterator is an object that implements the `__iter__()` and `__next__()` methods. Iterators allow you to traverse through all the elements of a collection.

```
class Counter:  
    def __init__(self, low, high):  
        self.current = low  
        self.high = high  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.current > self.high:  
            raise StopIteration  
        else:  
            self.current += 1
```

```
        return self.current - 1

for num in Counter(1, 3):
    print(num)
```

Decorators

Decorators are functions that modify the behavior of other functions. They are often used for logging, access control, and timing.

```
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Key Points:

- Decorators use the @decorator_name syntax.
- They help in code reuse and separation of concerns.

Dataclasses

Dataclasses, introduced in Python 3.7, provide a decorator and functions for automatically adding special methods such as `__init__()` and `__repr__()` to user-defined classes.

```
from dataclasses import dataclass

@dataclass
```

```
class Point:
    x: int
    y: int

p = Point(1, 2)
print(p) # Output: Point(x=1, y=2)
```

Benefits:

- Reduces boilerplate code
 - Provides built-in methods for comparison and representation
-

List Comprehension

List comprehensions offer a concise way to create lists using a single line of code.

```
squares = [x**2 for x in range(5)]
print(squares) # Output: [0, 1, 4, 9, 16]
```

Advantages:

- More readable and compact than traditional loops
 - Can include conditions: [x for x in range(10) if x % 2 == 0]
-

Dictionary Comprehension

Dictionary comprehensions allow you to construct dictionaries in a clear and concise way.

```
squares = {x: x**2 for x in range(5)}
print(squares) # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Use cases:

- Transforming or filtering dictionary data
- Creating mappings from sequences

`__slots__`

The `__slots__` declaration in a Python class is used to explicitly declare data members and prevent the creation of a dynamic `__dict__` for each instance. This can save memory and improve attribute access speed, especially when creating many instances.

```
class Person:
    __slots__ = ['name', 'age']

    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 30)
print(p.name)
```

Benefits:

- Reduces memory usage by avoiding per-instance dictionaries
- Prevents accidental creation of new attributes

Limitations:

- Cannot add attributes not listed in `__slots__`
- May not work well with multiple inheritance