Introduction to Python Cheat Sheet

by Arshdeep via cheatography.com/201979/cs/42848/

Looping Statements

For Loop	`for item in iterable:` `  # Code block to be executed for each item`
Example	`for i in range(5):` `  print(i)`
Output	0 1 2 3 4
While Loop	`while condition:` `  # Code block to be executed as long as condition is True`

Example	`count = 0`
	`while count < 5:`
	`  print(count)`
	`  count += 1`

Output	0 1 2 3 4
break	Terminates the loop immediately
continue	Skips the rest of the code inside the loop for the current iteration and proceeds to the next iteration
pass	Acts as a placeholder, does nothing

Nested Loops	`for i in range(3):`
	`  for j in range(2):`
	`  print(i, j)`

List Compre-hension	`[expre ssion for item in iterable]`
Dictionary Compre-hension	`{key_e xpr ession: value_ exp ression for item in iterable}`
Generator Expression	`(expre ssion for item in iterable)`

Evolution of Python

Birth of Python (1989-1991)	Created by Guido van Rossum, Python emerged in the late 1980s as a successor to the ABC language. Its name was inspired by Monty Python's Flying Circus, a British sketch comedy series.

Evolution of Python (cont)

Python 1.0 (1994)	Python 1.0 was released with features like lambda, map, filter, and reduce. Its simplicity and readability gained attention in the programming community.
Python 2.x Series (2000--2008)	Python 2 introduced significant improvements and became widely adopted. However, this series faced challenges with compatibility issues when Python 3 was released.
Python 3.x Series (2008-pre-sent)	Python 3 marked a major overhaul of the language, aiming to fix inconsistencies and introduce new features while maintaining backward compatibility. Despite initial resistance, it eventually gained widespread acceptance.
Python Enhanc-ement Proposals (PEPs)	PEPs serve as the formal mechanisms for proposing major changes to Python. They facilitate community discussion and decision-making processes, ensuring Python's evolution reflects the needs of its users.
Community and Ecosystem Growth	Python's open-source nature has fostered a vibrant community, contributing to a vast ecosystem of libraries, frameworks, and tools. This growth has propelled Python to become one of the most popular and versatile programming languages worldwide.

Evolution of Python (cont)

Recent Developments	Continual updates and enhancements keep Python relevant and competitive in the ever-changing landscape of programming languages. Recent developments include optimizations for performance, improvements in concurrency, and enhancements in data science and machine learning capabilities.
Future Directions	Python continues to evolve, with ongoing efforts to enhance performance, maintainability, and ease of use. The community-driven development model ensures that Python remains adaptable to emerging technologies and evolving programming paradigms.

Rules for Identifiers

Must start with a letter (a-z, A-Z) or underscore (_).
Can be followed by letters, digits (0-9), or underscores.
Python identifiers are case-sensitive.
Cannot be a reserved word or keyword.

Identity Operators

Is: is
Is not: is not

Membership Operators

In: in
Not in: not in

Data Types

Integer (int)	Represents whole numbers.
Float (float)	Represents floating-point numbers (decimal numbers).
String (str)	Represents sequences of characters enclosed in quotes (' or ").
Boolean (bool)	Represents truth values True or False.
List	Ordered collection of items, mutable.
Tuple	Ordered collection of items, immutable.
Dictionary (dict)	Collection of key-value pairs, unordered.

Data Types (cont)

Set	Collection of unique items, unordered.

Data Types

Integer (int)	Represents whole numbers.
Float (float)	Represents floating-point numbers (decimal numbers).
String (str)	Represents sequences of characters enclosed in quotes (' or ").
Boolean (bool)	Represents truth values True or False.
List	Ordered collection of items, mutable.
Tuple	Ordered collection of items, immutable.
Dictionary (dict)	Collection of key-value pairs, unordered.
Set	Collection of unique items, unordered.

Features of Python

Simple and Readable Syntax	Python's syntax is designed to be simple and readable, making it easy for beginners to learn and understand. Its clean and concise syntax reduces the cost of program maintenance.
Interpreted Language	Python is an interpreted language, meaning that it does not need to be compiled before execution. This allows for rapid development and testing of code.
High-Level Language	Python abstracts low-level details like memory management and provides constructs like objects, functions, and modules, allowing developers to focus on solving problems rather than dealing with system-level concerns.
Dynamic Typing	Python is dynamically typed, meaning you don't need to declare the data type of variables explicitly. This makes Python code shorter and more flexible.

Features of Python (cont)

Object-Oriented	Python supports object-oriented programming (OOP) paradigms, allowing developers to create reusable and modular code by defining classes and objects.
Extensive Standard Library	Python comes with a vast standard library that provides modules and functions for a wide range of tasks, from file I/O to networking to web development. This reduces the need for third-party libraries for many common tasks.
Cross-Platform Compatibility	Python code can run on various platforms such as Windows, macOS, and Linux without modification, making it highly portable.
Dynamic Memory Allocation	Python uses dynamic memory allocation and garbage collection, automatically managing memory usage and freeing up memory when objects are no longer needed.
Strong Community Support	Python has a large and active community of developers who contribute to its growth by creating libraries, frameworks, and tools. This vibrant community ensures that there are resources and support available for developers at all levels.
Integration Capabilities	Python can easily integrate with other languages like C/C++, allowing developers to leverage existing code and libraries written in other languages.

Features of Python (cont)

Ease of Learning and Deployment	Python's simplicity and readability make it an excellent choice for beginners, and its extensive documentation and community support make it easy to learn and deploy for both small-scale and large--scale projects.
Scalability	While initially known for its simplicity and ease of use, Python is also scalable and can handle large-scale projects effectively. With frameworks like Django and Flask for web development, and libraries like NumPy and Pandas for data science, Python is suitable for a wide range of applications, from small scripts to enterprise-level systems.

Comparison Operators

Equal to: ==

Not equal to: !=

Greater than: >

Less than: <

Greater than or equal to: >=

Less than or equal to: <=

Logical Operators

Logical AND: and

Logical OR: or

Logical NOT: not

Functions

Defining a Function

```
def function_name(parameters):
    """docstring"""
    # code block
    return value
```

Calling a Function

```
result = functi on_ nam e(a rgu ments)
```

Positional Parameters

```
def greet(name):
    print( " Hel lo, ", name)
greet( " Ali ce")  # Output: Hello, Al
ice
```

Functions (cont)

Keyword Parameters	`def greet( name, greeting):`
	`  print( gre eting, name)`
	`greet( nam e="B ob", greeting="Hi")`
	`# Output: Hi Bob`

Default Parameters	`def greet( name, greeting="Hello"):`
	`  print( gre eting, name)`
	`greet("Alice")`
	`# Output: Hello Alice`

*args (Non-keyword Arguments)	`def add(*args):`
	`  return sum(args)`
	`add(1, 2, 3)  # Output: 6`

**kwargs (Keyword Arguments)	`def details(**kwargs):`
	`  print(kwargs)`
	`detail s(n ame ="Al ice ", age=30)`
	`# Output: {'name': 'Alice', 'age': 30}`

Docstrings	`def function_name(parameters):`
	`  " " " Des cri ption of the functi on"""`
	`  # code block`
	`  return value`
	`print( fun cti on_ nam e._ _doc__)`

Return Statement	def add(a, b): return a + b result = add(3, 5) # Output: 8
Local Scope	Variables defined inside a function have local scope.
Global Scope	Variables defined outside functions have global scope.
Lambda Functions	`double = lambda x: x * 2` `print( dou ble(5))  # Output: 10`

Recursive Functions	`def factorial(n):`
	`  if n == 0:`
	`  return 1`
	`  else:`
	`  return n * factorial(n-1)`
	`print( fac tor ial(5))  # Output: 120`

Exception Handling

What is an Exception?	An exception is an error that occurs during the execution program. It disrupts the normal flow of the program's instr ctions.

try-except block	`try:`
	`  # Code that may raise an exception`
	`except ExceptionType:`
	`  # Code to handle the exception`

try-except-else block	`try:`
	`  # Code that may raise an exception`
	`except ExceptionType:`
	`  # Code to handle the exception`
	`else:`
	`  # Code to execute if no exception occ`

try-except-finally block	`try:`
	`  # Code that may raise an exception`
	`except ExceptionType:`
	`  # Code to handle the exception`
	`finally:`
	`  # Code that will execute no matter wh`

Built-in Exceptions	Examples include TypeError, ValueError, ZeroDivisionErr etc.
Raising Exceptions	`raise Except ion Typ e("Error messag e")`

Custom Exceptions	`class CustomError(Exception):`
	`  def __init __( self, message):`
	`  self.m essage = message`

Handling Multiple Exceptions	`try:`
	`  # Code`
	`except (Excep tio nType1, Except ion Type2 s e:`
	`  # Handle both exceptions`

File Handling

Opening a File	`file = open("f ile nam e.t xt", " r")`
Closing a File	`file.c lose()`
Reading from a File	`content = file.r ead()`
Writing to a File	`file.w rit e("H ello, world! ")`
Appending to a File	`file = open("f ile nam e.t xt", " a")` `file.write("New conten t")`
Iterating Over Lines	for line in file: print(line)
Checking File Existence	import os.path if os.path.exists("filename.txt"): print("File exists")

File Handling with Context Managers	`with open("f ile nam e.t xt", " r") as file:`
	`  content = file.read()`
	`  # file automa tically closed after exiting the 'with' block`

Arithmetic Operators

Addition: +
Subtraction: -
Multiplication: *
Division: /
Modulus: %
Exponentiation: **

Common Functions

print()	Output text or variables to the console.
input()	Receive user input from the console.
len()	Calculate the length of a sequence (e.g., string, list, tuple).
range()	Generate a sequence of numbers within a specified range.
type()	Determine the type of a variable or value.
int()	Convert a value to an integer.
float()	Convert a value to a floating-point number.
str()	Convert a value to a string.
list()	Convert a sequence (e.g., string, tuple) to a list.
tuple()	Convert a sequence (e.g., string, list) to a tuple.

Common Functions (cont)

dict()	Create a dictionary or convert a sequence of key-value pairs into a dictionary.
sorted()	Return a new sorted list from the elements of an iterable.
max()	Return the largest item in an iterable or the largest of two or more arguments.
min()	Return the smallest item in an iterable or the smallest of two or more arguments.
sum()	Return the sum of all elements in an iterable.
abs()	Return the absolute value of a number.
round()	Round a floating-point number to a specified precision.
zip()	Combine multiple iterables into tuples.
enumer-ate()	Return an enumerate object, which yields pairs of index and value.
map()	Apply a function to every item in an iterable.
filter()	Construct an iterator from those elements of an iterable for which a function returns true.
reduce()	Apply a rolling computation to sequential pairs of values in an iterable.
any()	Return True if any element of the iterable is true.
all()	Return True if all elements of the iterable are true.
dir()	Return a list of valid attributes for the specified object.
help()	Access Python's built-in help system.

Conditional Statements

If Statement	`if condition:` `  # Code to execute if condition is True`

If-else Statement	`if condition:`
	`  # Code to execute if condition is True`
	`else:`
	`  # Code to execute if condition is Fals`

Conditional Statements (cont)

If-elif-else Statement	`if condition1:`
	`  # Code to execute if condition1 is True`
	`elif condition2:`
	`  # Code to execute if condition2 is True`
	`else:`
	`  # Code to execute if all conditions are False`

Nested If Statements	`if condition1:`
	`  if condition2:`
	`  # Code to execute if both condition1 and condition2 are True`

Ternary Conditional Operator	`result = true_value if condition else false_value`

Short Circuit Evaluation	`# Example using 'and'`
	`if x > 0 and y < 10:`
	`  # Code here`
	
	`# Example using 'or'`
	`if a == 0 or b == 0:`
	`  # Code here`

Membership Test	`if item in list:` `  # Code to execute if item is present in list`
Identity Test	`if x is y:` `  # Code to execute if x and y refer to the same object`

Bitwise Operators

- Bitwise AND: &
- Bitwise OR: |
- Bitwise XOR: ^
- Bitwise NOT: ~
- Left shift: <<
- Right shift: >>

Assignment Operators

- Assign value: =
- Add and assign: +=
- Subtract and assign: -=
- Multiply and assign: *=
- Divide and assign: /=
- Modulus and assign: %=
- Exponentiation and assign: **=

Variables

Variable	Variables are used to store data values.
Variable Declaration	No explicit declaration needed. Just assign a value to a name.
Variable Naming	Follow naming conventions. Use descriptive names, avoid reserved words, and start with a letter or underscore.
Data Types	Variables can hold various data types such as integers, floats, strings, lists, tuples, dictionaries, etc.
Dynamic Typing	Python is dynamically typed, meaning you can reassign variables to different data types.

Example	`# Variable assignment`
	`x = 10`
	`name = " Ali ce"`
	`is_student = True`

Variable Reassi-gnment:	`x = 10 print(x)  # Output: 10`
	`x = " Hel lo"`
	`print(x)  # Output: Hello`

Multiple Assignment	`a, b, c = 1, 2, 3`
Constants	`PI = 3.14159`

Best Practices for Identifiers

- Use descriptive names for better code readability.
- Avoid using single letters or abbreviations that may be ambiguous.
- Follow naming conventions (e.g., snake_case for variables and functions, PascalCase for class names).

Tokens

Identifiers	These are names given to entities like variables, functions, classes, etc. They must start with a letter or underscore and can be followed by letters, digits, or underscores.
Keywords	Python has reserved words that have special meanings and cannot be used as identifiers. Examples include if, else, for, while, def, class, etc.
Literals	These are the raw data values used in a program. Common types of literals in Python include integers, floating-point numbers, strings, and boolean values.

Tokens (cont)

Operators	Operators are symbols used to perform operations on operands. Python supports various types of operators such as arithmetic operators (+, -, *, /), assignment operators (=, +=, -=), comparison operators (==, !=, <, >), logical operators (and, or, not), etc.
Delimiters	Delimiters are characters used to define the structure of a program. Examples include parentheses (), braces {}, square brackets [], commas ,, colons :, etc.
Comments	Comments are used to annotate code and are ignored by the Python interpreter. They start with the # symbol for single-line comments or are enclosed within triple quotes """ for multi-line comments.

Applications of Python

Web Development	Python's frameworks like Django and Flask are widely used for building web applications due to their simplicity and scalability.
Data Science	Python's libraries like NumPy, Pandas, and Matplotlib make it a preferred choice for data analysis, visualization, and machine learning tasks.
Artificial Intelligence and Machine Learning	Python provides extensive libraries such as TensorFlow, Keras, and PyTorch, making it popular for AI and ML projects.
Automation and Scripting	Python's ease of use and readability make it ideal for automating repetitive tasks and scripting.
Game Development	Python's simplicity and versatility are leveraged in game development frameworks like Pygame.

Applications of Python (cont)

Desktop GUI Applications	Libraries such as Tkinter and PyQt allow developers to create cross-platform desktop GUI applications easily.
Scientific Computing	Python is widely used in scientific computing for simulations, mathematical modeling, and data analysis in fields such as physics, engineering, and biology.
Finance and Trading	Python is extensively used in finance for tasks like algorithmic trading, risk management, and quantitative analysis due to its robust libraries and ease of integration.
Education	Python's readability and simplicity make it an excellent choice for teaching programming to beginners, as well as for educational software development.
Networking	Python's libraries like socket and Twisted are used for network programming, making it a popular choice for developing network-related applications.

Paradigms of Python

Imperative Programming	Focuses on describing how a program operates through a sequence of statements. Python's imperative style involves defining functions, loops, and conditional statements to control program flow.

Paradigms of Python (cont)

Object-Oriented Programming (OOP)	Emphasizes the creation of objects which encapsulate data and behavior. Python supports classes, inheritance, polymorphism, and encapsulation, enabling developers to structure their code in a modular and reusable manner.
Functional Programming	Treats computation as the evaluation of mathematical functions. Python supports functional programming concepts such as higher-order functions, lambda expressions, and immutable data structures. Functional programming encourages writing pure functions without side effects, enhancing code readability and testability.
Procedural Programming	Involves organizing code into procedures or functions to perform tasks. Python supports procedural programming by allowing the creation of functions and modules to break down tasks into smaller, manageable units.
Aspect-Oriented Programming (AOP)	Focuses on separating cross-cutting concerns such as logging, authentication, and error handling from the main program logic. While Python doesn't provide built-in AOP support, libraries like AspectLib and Pythoscope offer AOP capabilities through decorators and metaprogramming.

Modules

What are Modules?	Modules in Python are files containing Python code. They can define functions, classes, and variables. Python code in one module can be reused in another module.

Modules (cont)

Why Use Modules?	Encapsulation: Keep related code together for better organization. Reusability: Write code once and reuse it in multiple places. Namespacing: Avoid naming conflicts by using module namespaces.
Importing Modules	
Use the import keyword to import a module.	`import module _name`
Use from keyword to import specific items from a module.	`from module _name import item1, item 2`
Standard Library Modules	Python comes with a rich standard library of modules for various tasks. Examples: math, os, datetime, random, json, csv, etc.
Third-Party Modules	Extensive collection of third-party modules available via the Python Package Index (PyPI).
Install third-party modules using pip, the Python package manager.	`pip install module _name`
Creating Modules	To create your own module, simply save Python code in a .py file. Functions, classes, and variables defined in the file become accessible when the module is imported.
Special Attributes	__name__: Name of the module. When a module is run as a script, its __name__ is set to "__main__". __file__: Path to the module's source file.

Modules (cont)

Best Practices	Use meaningful names for modules. Document your modules using docstrings. Avoid polluting the global namespace by importing only what you need. Follow PEP 8 guidelines for code style.
Importing standard library module:	`import math` `print(math.pi)`
Importing specific items	`from math import pi, sqrt` `print(pi)`

Object Oriented Programming

Object-Oriented Programming (OOP) in Python	Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which can contain data (attributes) and code (methods). Python supports OOP principles, making it versatile and powerful for building complex systems.
Class	A class is a blueprint for creating objects.
Object	An object is an instance of a class, representing a specific entity in your program.
Encapsulation	Encapsulation refers to bundling data (attributes) and methods that operate on the data within a single unit, i.e., a class. It helps in data hiding and abstraction, enabling better control over data access and modification.

Object Oriented Programming (cont)

Inheritance	Inheritance allows a class (subclass/child class) to inherit attributes and methods from another class (superclass/parent class). It promotes code reuse and facilitates building hierarchical relationships between classes.
Polymorphism	Polymorphism enables a single interface to be used for different data types or objects. It allows methods to behave differently based on the object they are called on, promoting flexibility and extensibility.
Modularity	OOP promotes modular design, making code more organized and easier to maintain.
Reusability	Through inheritance and polymorphism, code reuse is facilitated, reducing redundancy.
Scalability	OOP supports building large-scale applications by structuring code into manageable units.
Abstraction	OOP allows developers to focus on high-level functionality without worrying about implementation details.