Python

Python Fundamentals

COMPLETE NOTES

Variables - Operators - Control Flow - Loops - Functions - Lists - Dictionaries - Strings - Input - OOP

Learn With Asad

learnwithasad.com

Contents

01 Getting Started

What is Python? Why print()?

What is Python?

Python is one of the world's most popular programming languages, created by **Guido van Rossum** in 1991. It is used for:

- Artificial Intelligence and Machine Learning
- Web Development (Django, Flask)
- Data Science and Analysis
- Automation and Scripting

Python reads almost like **plain English**, making it ideal for beginners and professionals alike.

Your First Program -- print()

```
# This is your first Python program
print("Hello, World!")
```

The print() function displays text on the screen. In the 1950s-60s, computers had no screens -- output was physically **printed** on paper via a teletype machine. The name stuck through generations of programming languages.

```
print("My name is Asad")  # text
print(42)               # numbers -- no quotes needed
print(3.14)
print(True)
```

Pro Tip: Each print() call starts on a new line automatically.

Comments

Comments are notes in code that Python completely **ignores** -- they exist only for humans.

```
# Single-line comment
x = 10  # inline comment

"""
Multi-line comment block
Describe the whole program here
"""

# TODO: Add a leaderboard here
```

Pro Tip: Great comments explain WHY the code does something, not WHAT it does.

02 Variables & Data Types

Labelled boxes in memory

Variables

A variable is a **labelled box in memory** that holds a value. Use = to assign.

```
player_name  = "Asad"   # str
player_score = 1500    # int
battery      = 87.5   # float
is_online    = True   # bool
guild_name   = None   # None
```

Watch Out: = is ASSIGNMENT (stores a value). == is COMPARISON (asks: are they equal?).

Data Types

Type	Description	Example
`int`	Whole numbers, no decimal	`42, -7, 0, 1000000`
`float`	Numbers with a decimal point	`3.14, -0.5, 99.99, 1.0`
`str`	Text -- any characters in quotes	`"Hello", 'Asad', "XK-47"`
`bool`	Only True or False (capital first!)	`True, False`
`None`	Represents no value / empty	`winner = None`

Dynamic Typing

Python figures out the type **automatically** from the value -- you don't declare types like in Java or C++.

```
score = 1500      # Python sees integer
name  = "Jordan" # Python sees string
score = "reset"  # can even change type later!
```

None -- The Empty Box

```
high_score  = None  # game hasn't been played yet
winner_name = None  # no winner determined
item        = None  # player hasn't chosen
```

Pro Tip: None is NOT zero, empty string, or False. It means the variable has absolutely no value.

Naming Conventions -- Snake Case

```
# Good naming (snake_case)
player_name  = "Asad"
total_score  = 1500
ticket_price = 12.50

# Bad naming
x = "Asad"         # unclear
3_lives = 3        # SyntaxError: starts with number
player score = 0   # SyntaxError: space in name
```

03 Operators

Arithmetic - Comparison - Logical

Arithmetic Operators

```
10 + 4   # 14   addition
10 - 4   # 6    subtraction
10 * 4   # 40   multiplication
10 / 4   # 2.5  division -- ALWAYS float!
10 // 4  # 2    floor division (whole number)
10 % 4   # 2    modulo (remainder)
2 ** 5   # 32   power / exponent
```

Watch Out: Division / ALWAYS returns a float: 10/2 gives 5.0 not 5. Use // for integer result.

The Modulo Operator %

Modulo gives the **remainder** after division. Incredibly useful in real programming.

```
10 % 3  # 1  (10 / 3 = 3 remainder 1)

# Even or Odd:
score = 48
if score % 2 == 0:
    print("Even")

# Items left over after packing:
chocolates = 17
box_size   = 5
leftover   = chocolates % box_size  # 2
```

Compound Assignment Operators

```
lives = 3
lives += 2   # lives = lives + 2  -> 5
lives *= 2   # lives = lives * 2  -> 10
lives -= 1   # lives = lives - 1  -> 9
lives //= 2  # lives = lives // 2 -> 4
lives %= 3   # lives = lives % 3  -> 1
lives **= 3  # lives = lives ** 3 -> 1
```

Pro Tip: You'll see += constantly in loops and score trackers -- it's the go-to shortcut.

Comparison Operators

Comparison operators always return **True** or **False**.

```
5 == 5    # True   (equal to)
5 != 3    # True   (not equal to)
10 > 7    # True   (greater than)
3 < 8     # True   (less than)
5 >= 5    # True   (greater or equal)
4 <= 6    # True   (less or equal)
```

Logical Operators

```
has_ticket = True
is_adult   = True
is_banned  = False
```

```
can_enter = has_ticket and is_adult and not is_banned
# True and True and not False -> True

# and: BOTH must be True
# or:  AT LEAST ONE must be True
# not: FLIPS True -> False, False -> True
```

Pro Tip: Split complex conditions into named variables first -- it makes code far easier to read.

04 Control Flow

if - elif - else - Nested conditions

The if Statement

The if statement lets your program make decisions -- run different code depending on a condition.

```
age = 15

if age >= 18:
    print("Adult")   # runs when True

print("Always runs")  # no indent = outside block
```

Watch Out: Never forget the colon : after the condition! Indent with 4 spaces.

if-else Statement

```
pocket = 12

if pocket >= 10:
    print("Buy the toy!")
else:
    print("Not enough money.")
```

if-elif-else Chain

```
marks = 72

if marks >= 90:
    grade = 'A'
elif marks >= 75:
    grade = 'B'
elif marks >= 60:
    grade = 'C'
else:
    grade = 'F'

print(grade)  # C
```

Pro Tip: Python checks from top to bottom. The FIRST True condition wins -- all others are skipped. ORDER MATTERS!

Nested if-else

```
age = 18

if age >= 18:
    title = "Adult"
    if age >= 21:
        can_drink = True
    else:
        can_drink = False
else:
    title = "Minor"
```

05 Loops
for - while - break - continue

The for Loop

A for loop repeats a block of code for each item in a sequence.

```
# Loop over a range of numbers
for i in range(5):        # 0, 1, 2, 3, 4
    print(i)

for i in range(1, 6):     # 1 to 5
    print(i)

for i in range(0, 10, 2): # 0, 2, 4, 6, 8 (step of 2)
    print(i)
```

```
# Loop over a list
subjects = ["Maths", "Science", "English"]
for subject in subjects:
    print(f"Studying: {subject}")

# With index:
for i, subject in enumerate(subjects):
    print(f"{i}: {subject}")
```

The while Loop

A while loop keeps running as long as a condition is True.

```
count = 0

while count < 5:
    print(f"Count: {count}")
    count += 1

# Output: Count 0, Count 1, Count 2, Count 3, Count 4
```

Watch Out: Always update the loop variable inside a while loop! Forgetting it causes an INFINITE LOOP.

break and continue

```
# break -- exit the loop immediately
for i in range(10):
    if i == 5:
        break         # stops here
    print(i)          # prints 0 1 2 3 4

# continue -- skip to next iteration
for i in range(5):
    if i == 2:
        continue      # skips 2
    print(i)          # prints 0 1 3 4
```

06 Functions

Define once, call many times

Defining and Calling Functions

```
# Define
def greet(name):
    message = f"Hello, {name}!"
    return message

# Call
result = greet("Asad")
print(result)   # Hello, Asad!
```

Functions with Multiple Parameters

```
def calculate_grade(marks, bonus=0):
    total = marks + bonus
    if total >= 90:
        return 'A'
    elif total >= 75:
        return 'B'
    else:
        return 'C'

print(calculate_grade(80, 12))  # A
print(calculate_grade(70))      # C (bonus=0 default)
```

Practical Example

```
def get_average(marks_list):
    total = sum(marks_list)
    return total / len(marks_list)

scores = [85, 92, 78]
print(get_average(scores))  # 85.0
```

Pro Tip: Functions help you avoid repeating code. If you copy-paste the same logic twice, make it a function!

07 Lists

Ordered, mutable collections

Creating & Accessing Lists

```
scores = [85, 92, 78, 61, 45]

scores[0]    # 85  (first item -- index 0)
scores[1]    # 92
scores[-1]   # 45  (last item)
scores[-2]   # 61  (second from end)

# Slicing
scores[1:3]  # [92, 78]  -- index 1 up to but not 3
scores[:3]   # [85, 92, 78]  -- first 3
scores[2:]   # [78, 61, 45]  -- from index 2 onwards
```

List Methods

```
scores = [85, 92, 78]

scores.append(61)      # add to END
scores.insert(0, 100)  # add at specific index
scores.remove(78)      # remove by VALUE
scores.pop()           # remove & return LAST item
scores.sort()          # sort ascending (in-place)
scores.reverse()       # reverse in-place
scores.sort(reverse=True)  # sort descending

len(scores)            # count of items
sum(scores)            # total of all numbers
max(scores)            # largest
min(scores)            # smallest
```

Looping Through Lists

```
fruits = ["apple", "banana", "mango"]

for fruit in fruits:
    print(fruit)

# With index:
for i, fruit in enumerate(fruits):
    print(f"{i}: {fruit}")
```

Pro Tip: Lists store items by POSITION. Use a dictionary when items need meaningful labels.

08 Dictionaries

Key-value pairs for labelled data

Creating & Accessing Dictionaries

```
player = {
    "name":     "Asad",
    "level":    12,
    "health":   100.0,
    "is_online": True
}

player["name"]    # "Asad"  (access by key)
player["level"]   # 12
```

Modifying Dictionaries

```
player["level"] = 13      # update existing key
player["coins"] = 500    # add new key-value pair
del player["is_online"]  # remove a key
print(player)
```

Dictionary Methods

```
student = {"name": "Riya", "age": 14, "grade": "A"}

student.keys()              # dict_keys(['name','age','grade'])
student.values()            # dict_values(['Riya', 14, 'A'])
student.items()             # key-value pairs

# Safe access -- won't crash if key missing:
student.get("score", 0)   # returns 0 if 'score' doesn't exist

# Check if key exists:
"name" in student           # True
"xp" in student             # False
```

Looping Through a Dictionary

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

# Output:
# name : Riya
# age  : 14
# grade: A
```

Pro Tip: Always use .get() instead of direct bracket access when you're not 100% sure the key exists.

09 Strings & f-Strings

Text manipulation and formatting

String Basics

```
name  = "Asad"           # double quotes
city  = 'Tokyo'          # single quotes -- both work!
msg   = "It's a great day!"  # apostrophe inside
multi = """Line 1
Line 2"""             # triple quotes for multi-line
```

String Operations

```
# Concatenation
full_name = first_name + " " + last_name

# Repetition
print("*" * 5)    # *****
print("-" * 30)   # 30 dashes (divider line)
```

f-Strings -- Modern Formatting (Python 3.6+)

```
name  = "Riya"
score = 92

# Old way (messy):
print("Well done " + name + "! Score: " + str(score))

# Modern f-string:
print(f"Well done, {name}! Score: {score}")

# Expressions inside {}:
print(f"Total: {25 * 4}")           # Total: 100
print(f"Pi: {3.14159:.2f}")         # Pi: 3.14 (2 decimal places)
print(f"CAPS: {name.upper()}")      # CAPS: RIYA
print(f"Big? {score > 80}")         # Big? True
```

String Methods

```
msg = "  Hello, Python World!  "
msg.upper()                 # UPPERCASE
msg.lower()                 # lowercase
msg.strip()                 # "Hello, Python World!"
msg.replace("Python","Amazing")  # replace text
msg.startswith("  Hello")    # True
msg.find("World")           # index position
len("hello")                # 5
"a,b,c".split(",")          # ['a', 'b', 'c']
"hello world".title()       # "Hello World"
```

Pro Tip: Always prefer f-strings when combining text and variables -- they handle type conversion automatically.

10 User Input

Making programs interactive

The input() Function

input() pauses the program, shows a prompt, and waits for the user to type and press Enter.

```
name = input("What is your name? ")
print(f"Hello, {name}! Welcome to Python!")
```

The Most Important Rule

Watch Out: input() ALWAYS returns a STRING -- even if the user types a number! You MUST convert it.

```
age_text = input("How old are you? ")
# type(age_text) ->

# This adds strings, not numbers:
age_text = "15"
print(age_text + age_text)  # "1515" -- string join!
```

Converting Input -- int() and float()

```
# Convert as you receive input (clean, one-liner):
age   = int(input("How old are you? "))     # str -> int
price = float(input("Enter price: "))        # str -> float

# Full interactive example:
name  = input("Name: ")
marks = int(input(f"Hello {name}! Enter marks: "))

if marks >= 75:
    print(f"Great job, {name}! Distinction!")
elif marks >= 40:
    print(f"Well done, {name}! You passed.")
else:
    print(f"Keep going, {name}!")
```

Pro Tip: Always tell the user what kind of input you expect. input("Enter age: ") is good. input(":") is confusing.

11 Object-Oriented Programming

Classes - Objects - Inheritance - Methods

What is OOP?

OOP models code like the real world -- as objects that have **properties** (attributes) and **actions** (methods).

A **class** is the blueprint (like a dog's description). An **object** is a real instance (like your actual dog, Buddy).

Your First Class

```
class Student:                       # PascalCase class name!
    def __init__(self, name, age):  # constructor
        self.name = name             # attribute
        self.age  = age

    def introduce(self):            # method
        print(f"Hi! I'm {self.name}, age {self.age}.")

# Create objects:
s1 = Student("Asad", 15)
s2 = Student("Riya", 14)

s1.introduce()     # Hi! I'm Asad, age 15.
print(s2.name)     # Riya
```

Inheritance

```
class Animal:               # parent class
    def __init__(self, name, age):
        self.name = name
        self.age  = age

    def breathe(self):
        print(f"{self.name} is breathing")

class Dog(Animal):          # child class -- inherits from Animal
    def bark(self):
        print(f"{self.name}: WOOF!")

dog = Dog("Bruno", 3)
dog.breathe()   # Inherited from Animal!
dog.bark()      # Dog's own method
```

Pro Tip: Use inheritance when the 'is-a' test passes: A Dog IS AN Animal. A Car IS AN Engine? No.

super() and Method Overriding

```
class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)  # run parent __init__ first
        self.breed = breed           # then add Dog's extras

    def speak(self):                 # override Animal's speak
        print(f"{self.name}: WOOF!")

class Cat(Animal):
```

```
    def speak(self):
        print(f"{self.name}: MEOW!")  # different behaviour!

# Polymorphism -- same method, different results:
for animal in [Dog("Bruno",3,"Lab"), Cat("Kitty",2)]:
    animal.speak()
```

__str__ -- Make Objects Print Nicely

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

    def __str__(self):
        return f"Student({self.name}, Marks: {self.marks})"

s = Student("Asad", 88)
print(s)   # Student(Asad, Marks: 88)
```

Pro Tip: Without __str__, printing an object shows its memory address like <__main__.Student object at 0x...>