# Python: Day 01

Introduction

# Objectives

## Foster a Strong Foundation

Understand the fundamental components and how to use them correctly

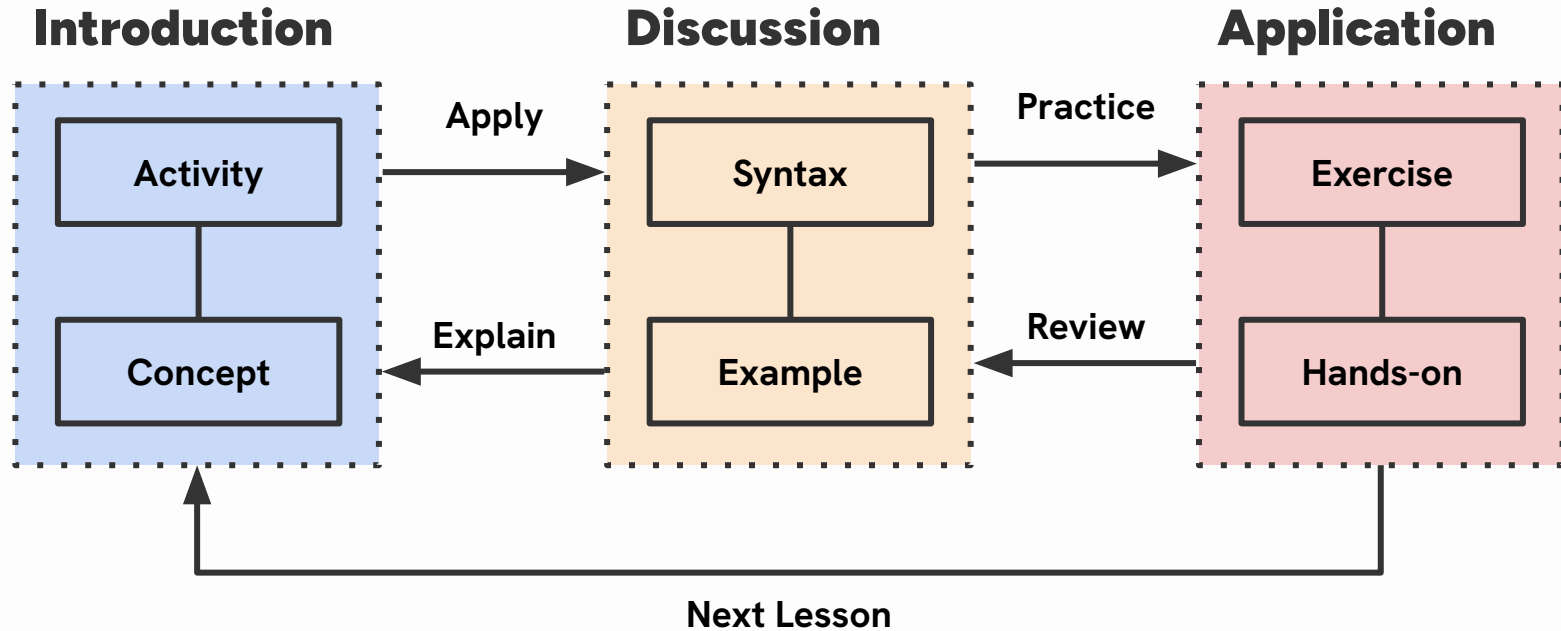## Develop Problem Solving Skills

Gain practical experience through activities, exercises, and lab sessions

## Prepare for Specialization

Provide a preview on how Python can be used in various contexts

# Structure

# Agenda

**01**

## Introduction

What is Python?

**02**

## Basics

Input-Output Process

**03**

## Control Flow

Logical Pathways

**04**

## Functions

Code Shortcuts

**05**

## Error Handling

Exceptional Cases

**06**

## Lab Session

Culminating Activity

# 01

# Introduction

Overview of Python and its potential

# Key Features

**Convenient**

Simple and concise
for easier development

**Modern**

Constantly updated with
useful features

**Active**

Large community with a
rich ecosystem

## Java

```
1  class HelloWorld {
2      public static void main(String args[]) {
3          System.out.println("Hello, World");
4      }
5  }
```

## C++

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello World" << std::endl;
5  }
```

# Python

```python
1  print("Hello World")
```

# Python Package Index (pypi.org)



Find, install and publish Python packages
with the Python Package Index

Search projects

Or browse projects

642,013 projects    7,001,441 releases    14,411,118 files    929,410 users

# Python Growth



Top programming languages on GitHub

RANKED BY COUNT OF DISTINCT USERS CONTRIBUTING TO PROJECTS OF EACH LANGUAGE.

1. Python
2. JavaScript
3. TypeScript
4. Java
5. C#
6. C++
7. PHP
8. Shell
9. C
10. Go

Obj-C   Ruby

2014  2016  2018  2020  2022  2024



Top 10 fastest growing languages in 2024

TAKEN BY PERCENTAGE GROWTH OF CONTRIBUTORS ACROSS ALL CONTRIBUTIONS ON GITHUB.

Python
TypeScript
Go
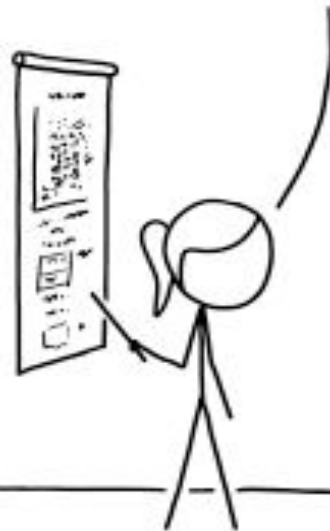HCL
Kotlin
Dart
Rust
Luna
TSQL
Blade

0   2M   4M   6M

2023   2024
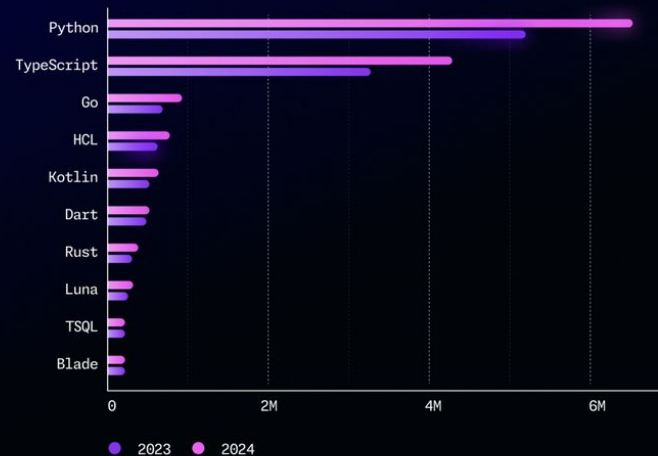
# Where can you use Python?

# Data Science

Python is famous for data science due to its mature yet widespread libraries on this field




xkcd

# Machine Learning

Math-intensive processes in machine learning are often made in low-level languages and interfaced with Python

# Web Development
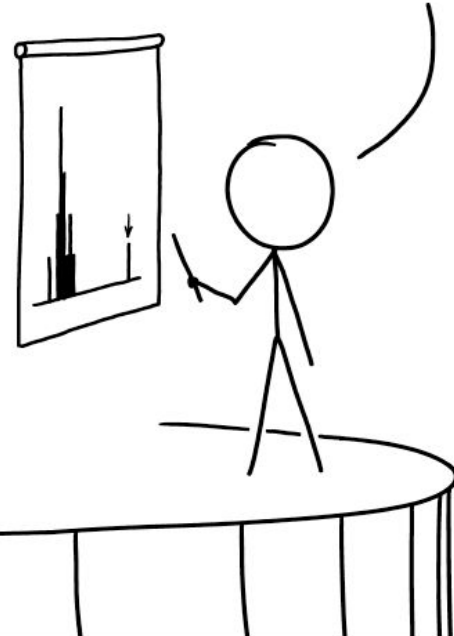
Alternatives to the traditional web tech stack include libraries and frameworks that Python can also provide

# Automation

Python gained its initial footing as an easy to use tool to automate mundane tasks
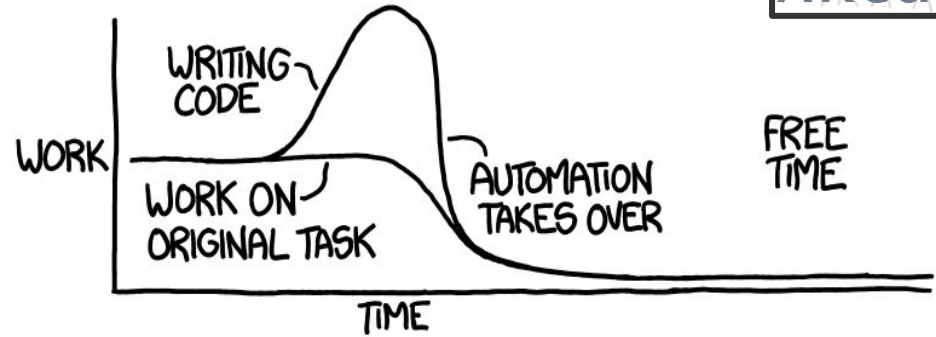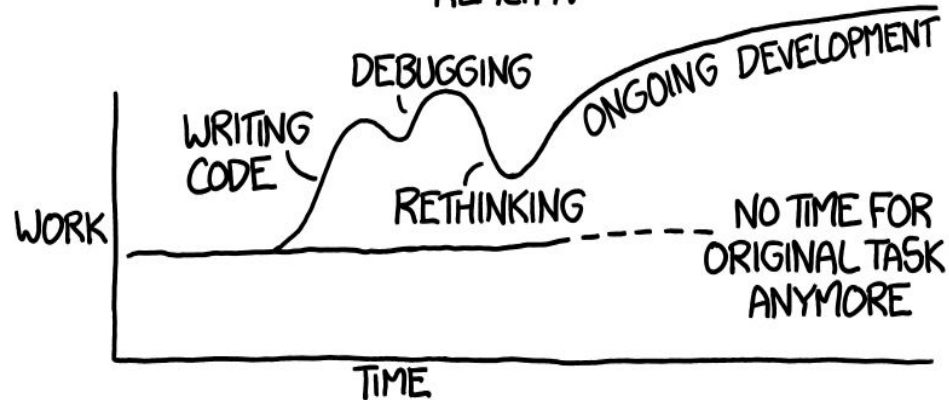
# Specialist Fields

Python's simplified syntax makes adoption to specialized fields much easier compared to other languages
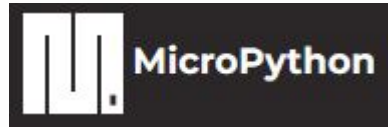
# Python History

## Origins

Python was created by Guido van Rossum in 1991 and released in 1994 (version 1.0) when was working the ABC Programming Language Group at the National Research Institute for Math and Computer Science in the Netherlands.



### Fun Fact # 1

The name Python was inspired by the BBC's TV Show: Monty Python's Flying Circus

### Fun Fact #2

Java's first version was released in 1995 by James Gosling, making Python older.

# Python History

### Python 1.x

Development started in 1991, but was officially released in January 1994. It was a part of Rossum's Computer Programming for Everybody (CP4E) initiative.

### Python 2.x

First instance released in October 2000, under a new license (Python Software Foundation License). This has been deprecated since January 2020.

### Python 3.x

First version released in December 2008, made to upgrade performance, add extra features, and improve clarity without being backwards compatible

# Setup Tools

Full instructions for your initial development setup

# Step 1: Download Git

Go to **https://git-scm.com/** and select the download option
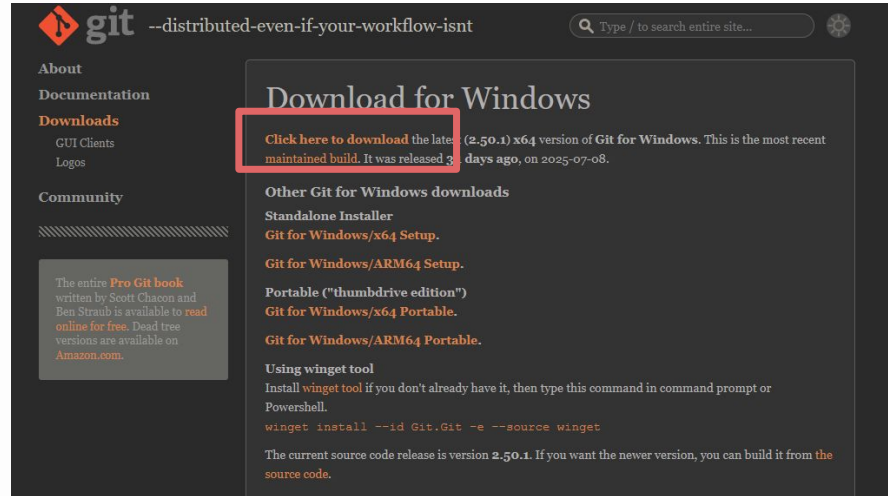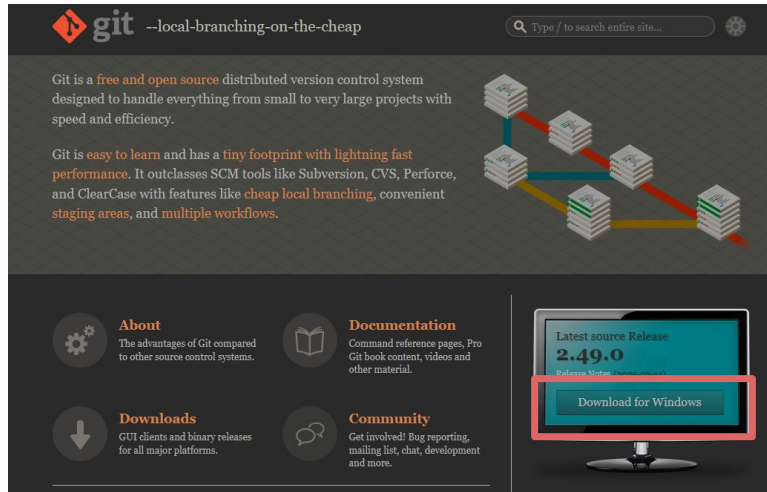
# Step 2: Git Installation Setup

Run the git installer and use the default options

# Step 3: Download Python Downloader

Go to **https://www.python.org/downloads/** and click the first download button
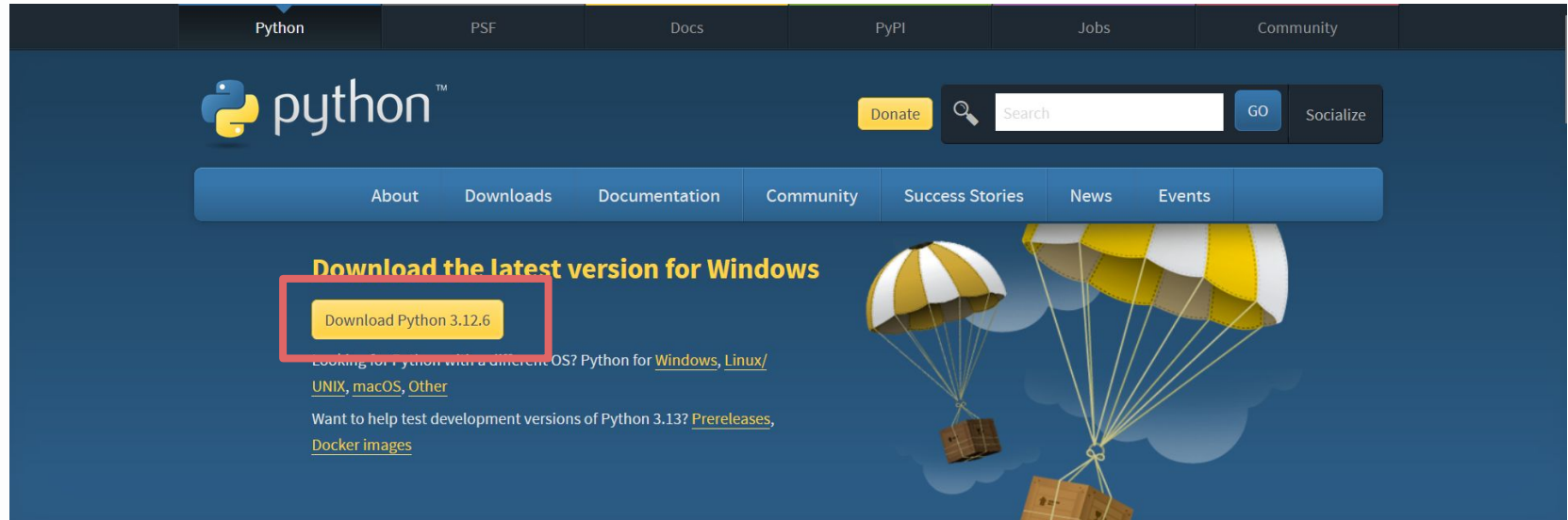
# Step 4: Run Python Installer

Run the downloaded installer. Select all the checkbox options and select *Install Now*.
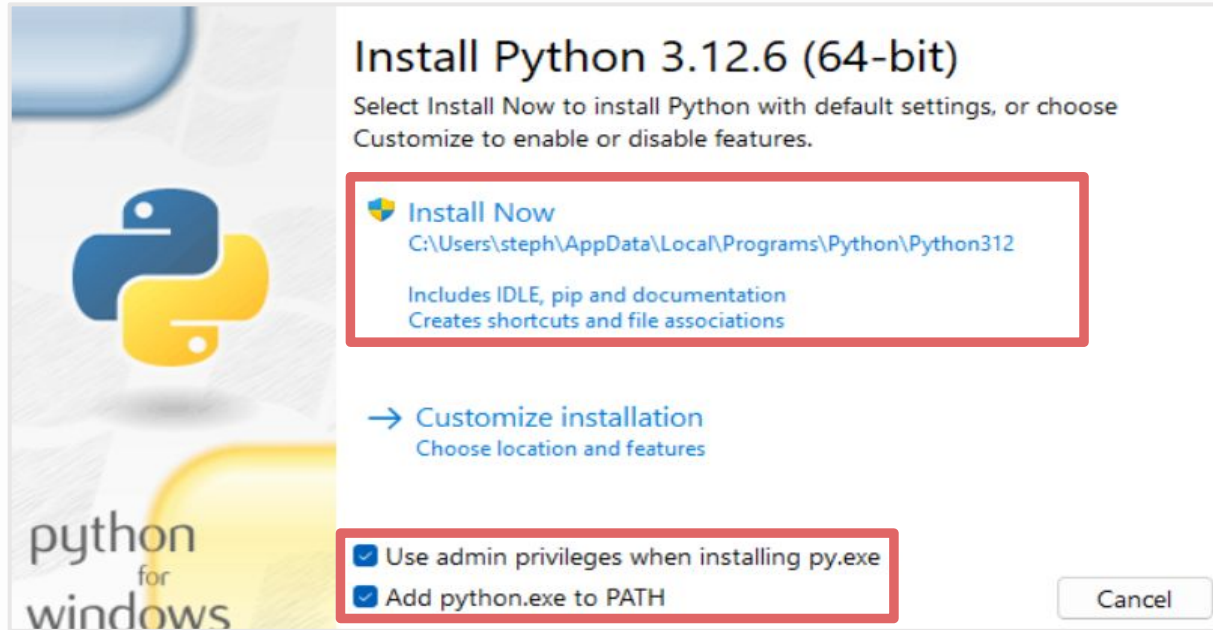
# Step 5: Download and Run PyCharm Installer

Go to **https://www.jetbrains.com/pycharm/download/** select *Download*.
Afterwards, run the installer and use the default options.

# Repository

Securing access to the course materials

# Repository System

| | Fork | |
|---|---|---|
| **Original** | → | **Online Copy** |
| | Full Online Copy | |

Online

- - - - - - - - - - - - - - - - - - - - - - - - - - - -

Local

Clone
(Download)

**Local Copy**
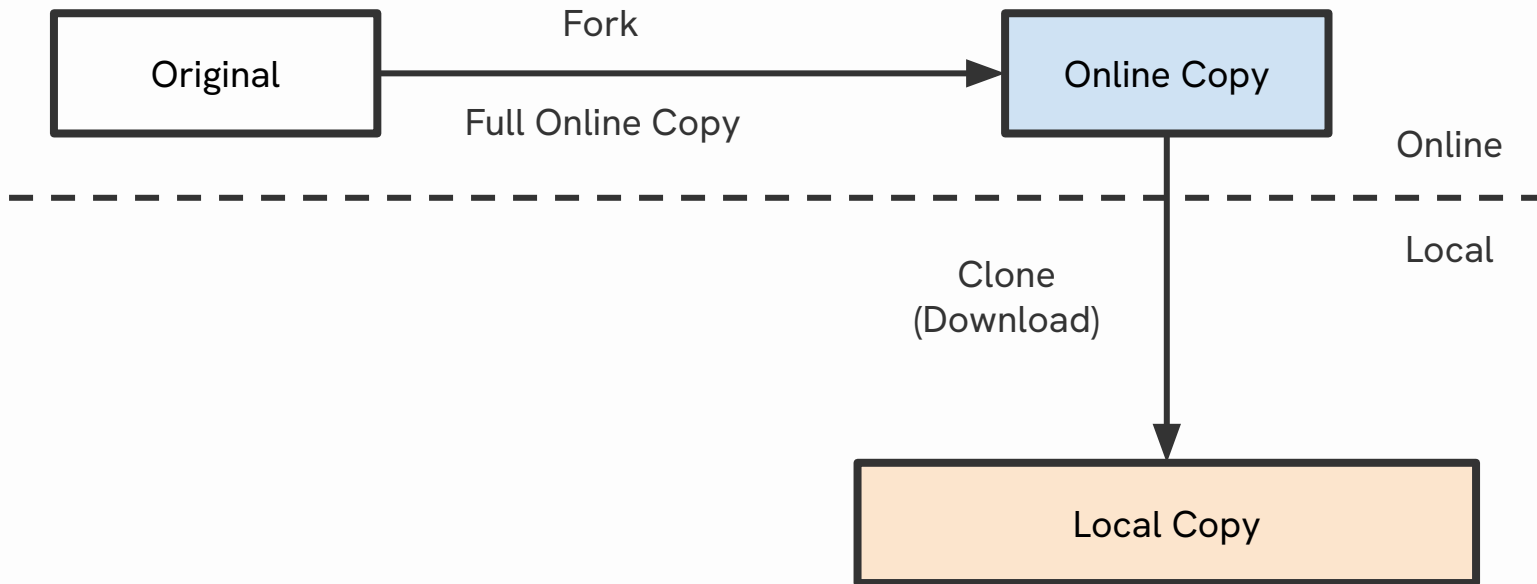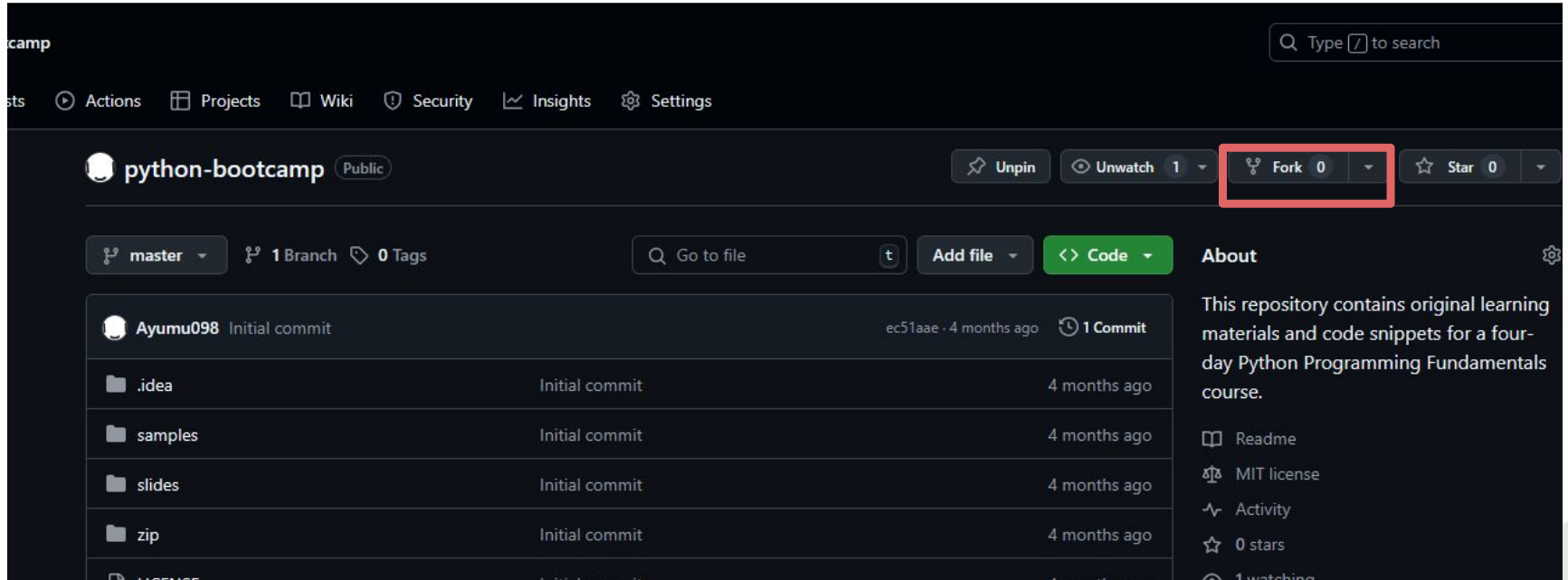
# Step 1: Setup Github Account

Go to **github.com** and *Sign In* if you already have an account or *Sign Up* for a new account
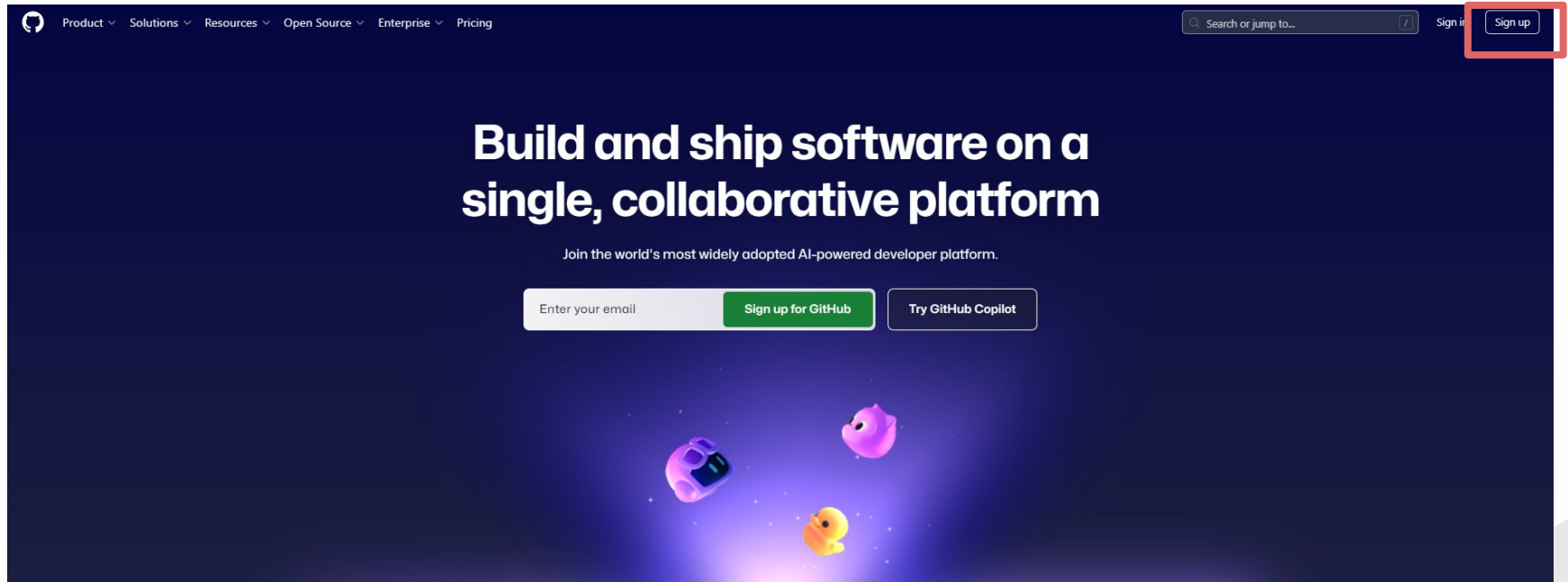
# Step 2: Fork python-bootcamp repository

Go to **tinyurl.com/pyboot** and select the *Fork* button on the upper right

# Step 3: Setup Fork Settings

Select the **Create Fork** button

# Step 4: Clone Repository

Run PyCharm Community and select the *Clone Repository*
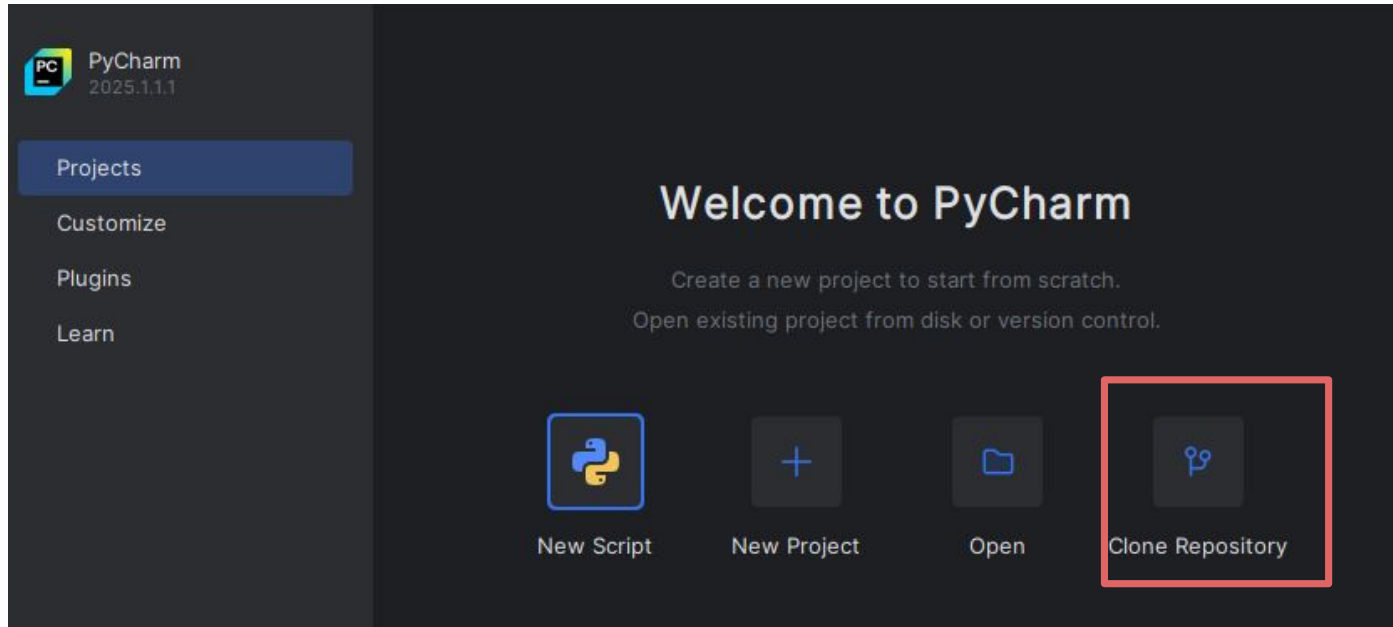
# Step 5: Login GitHub for PyCharm

On prompt, select *GitHub* on the upper left and *Log In via GitHub*. This will open the browser.
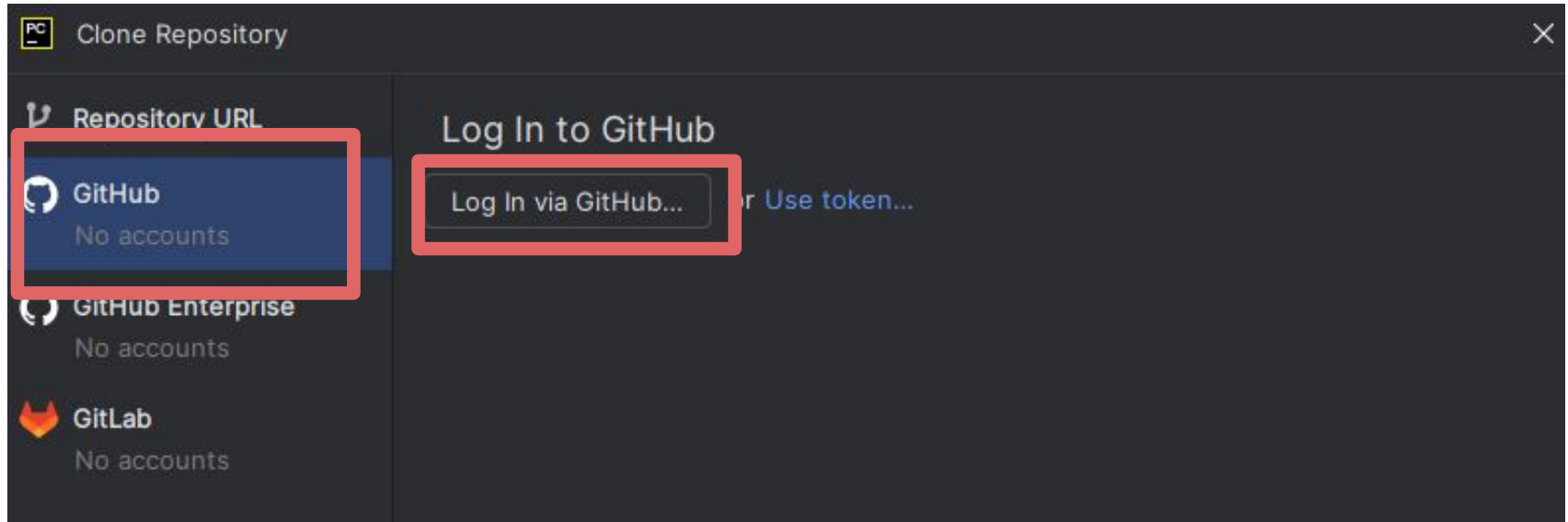
# Step 6: Authorize GitHub - PyCharm connection

A new tab will open in the browser. Select *Authorize in Github*

# Step 7: Select Repository

The window shows all your repositories. Select **python-bootcamp** and **Clone**.
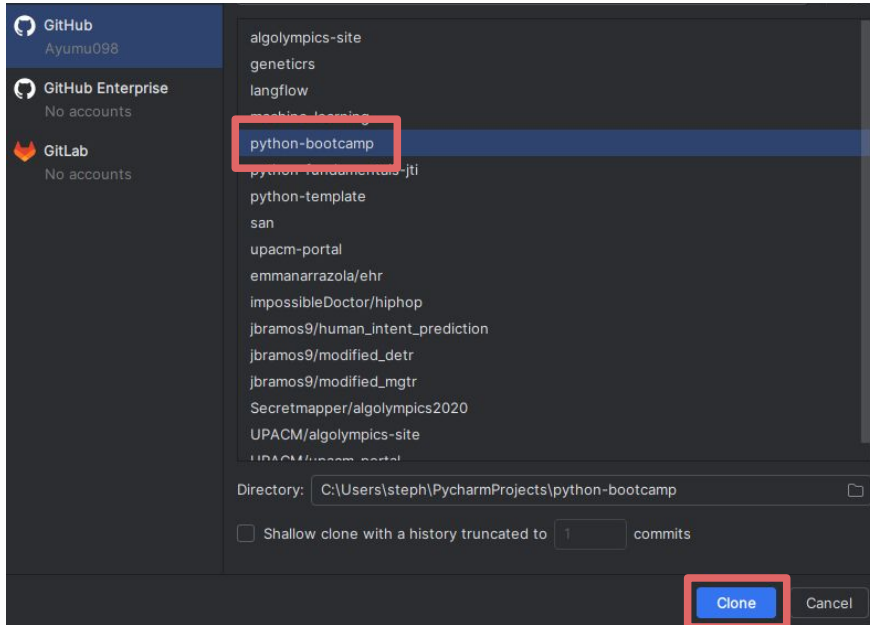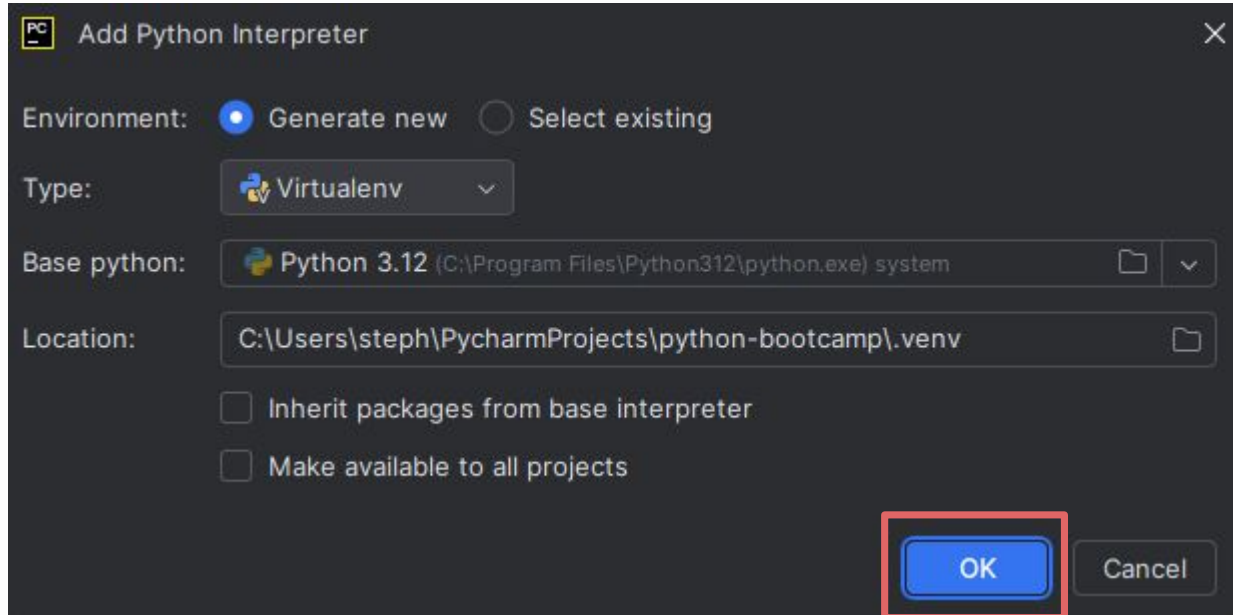
# Step 8: Add Python Interpreter

Go to *sample.py*. A yellow warning will appear when the file is opened. Select Configure Python Interpreter, Add New Interpreter, and Add Local Interpreter
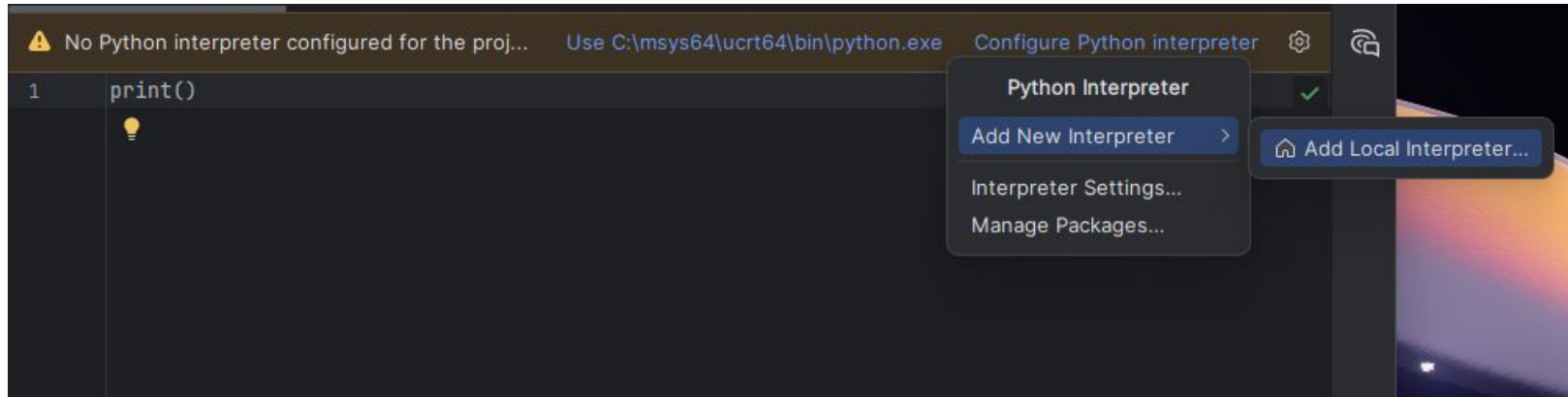
# Step 9: Setup Python Interpreter

On prompt, double check if Base Python is already setup and select **OK**.

# Hello World

A journey of a thousand miles begins with a single step

# Create a New Folder

Right click the current project folder name, select **New > Directory**.
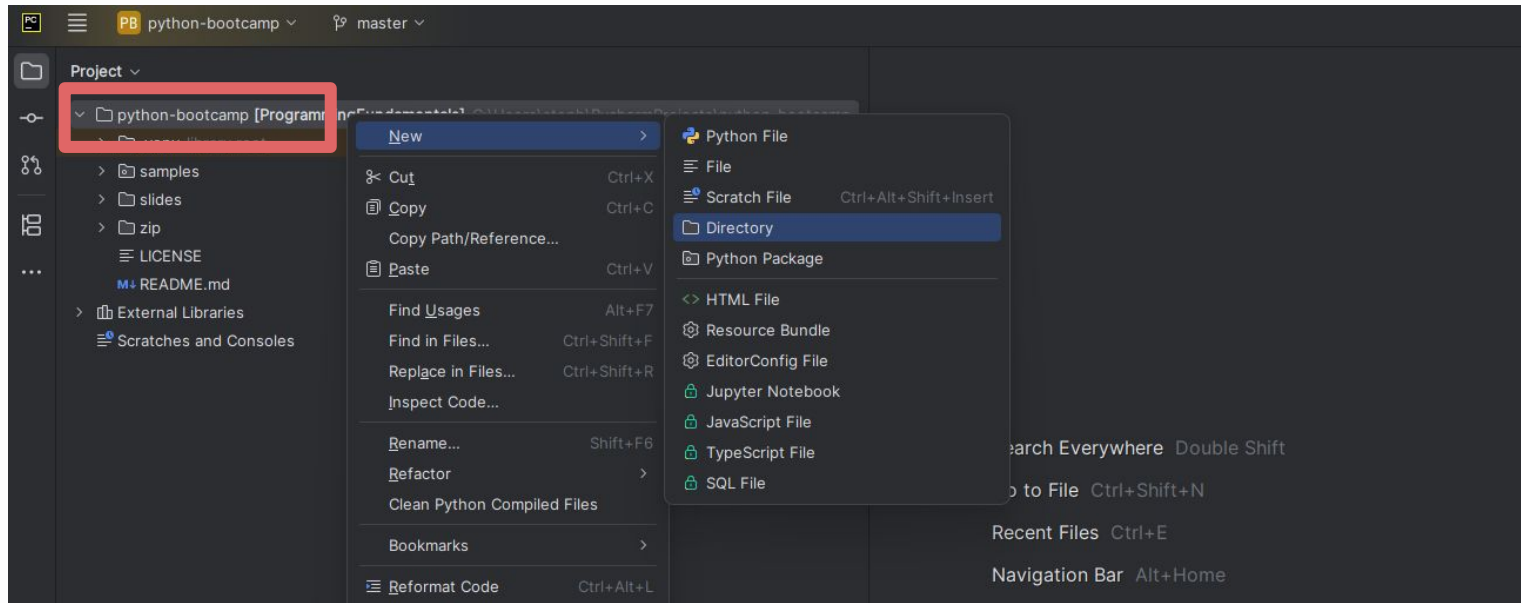
# Create New Python File

Right click the new folder name, select `New > Python File`.
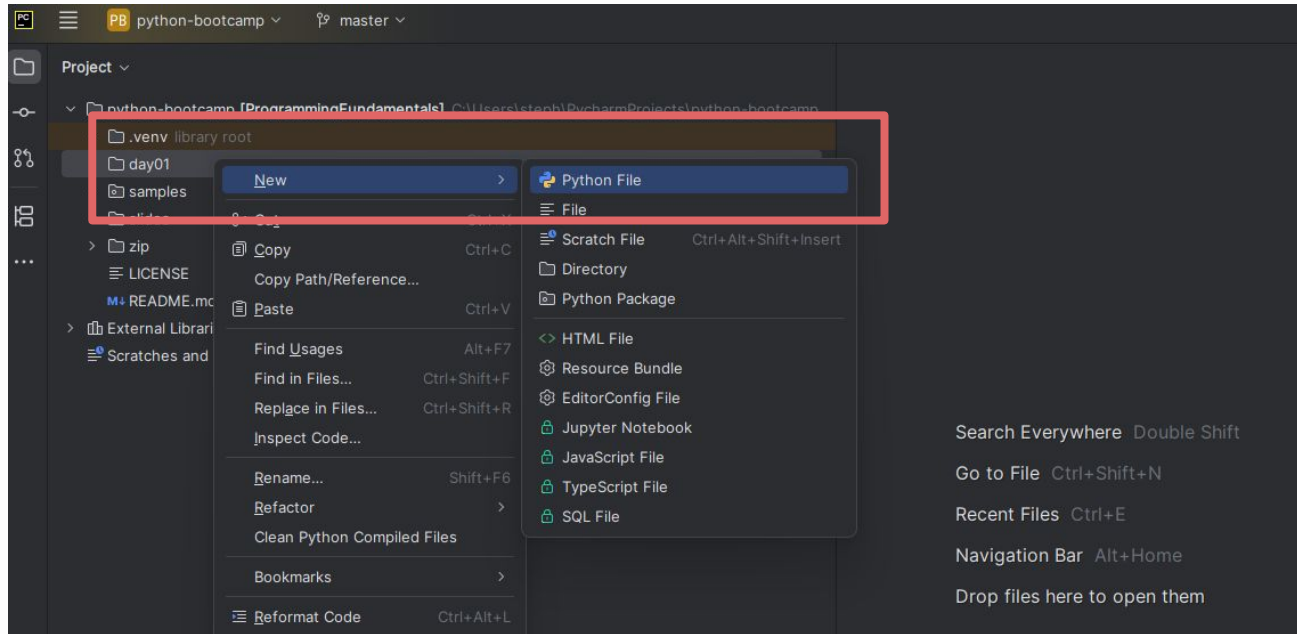
# Writing your First Code

## print ()

| Function | Predefined commands or actions |
| --- | --- |
| Parentheses | Marker where `function input` starts and ends |

# Writing your First Code

**print** (**Hello World** )

| | |
|---|---|
| **Function** | Predefined commands or actions |
| **Parentheses** | Marker where `function input` starts and ends |
| **Text** | |

# Writing your First Code

**print ( "Hello World" )**

| | |
|---|---|
| **Function** | Predefined commands or actions |
| **Parentheses** | Marker where `function input` starts and ends |
| **Text** | |
| **Double Quote** | Marker where the `text` starts and ends |

# Multi-line Printing

```
print ("Hello World")
print ("Hello Again!")
```

## Single-line Printing

```python
print ( "I" , "am" , "happy" )
```

# Quick Exercise: Hello World

Print the following in the console:

```
Hello! My name is your name
I am learning Python
```

# Comments for Documentation

Comments are usually used to describe, explain, or justify code

```python
1  # Practice for printing in multiple lines
2  print("Hello, I am new to Python")
3  print("Let's learn together!")
```

# Multiple-Line Comment

To write multiple lines without being detected as code, use triple quotes at the start and end

```
1  """
2  This is a very simple application to test the print function.
3  Remember that strings need double or single quotes!
4  """
5  print("Hello, I am new to Python")
6  print("Let's learn together!")
```

# Basics

Fundamental processes for data handling

# Variable Declaration

A variable can be created by writing its name, the equal sign, and the value

```
message = "Test Message"
```

message
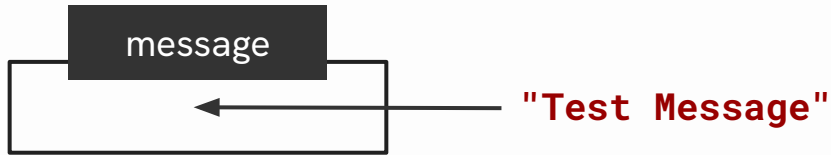
"Test Message"

# Variable Printing

Variables can also be displayed on the console with the **print** function

```
1   message = "Test Message"
```

```
2   print(message)
```

# Variables and Text

Be careful not to confuse strings and variables (no quotes)

```
1   message = "Test Message"
```

```
2   print(message)
```

≠

```
1   print("message")
```

Test Message

message
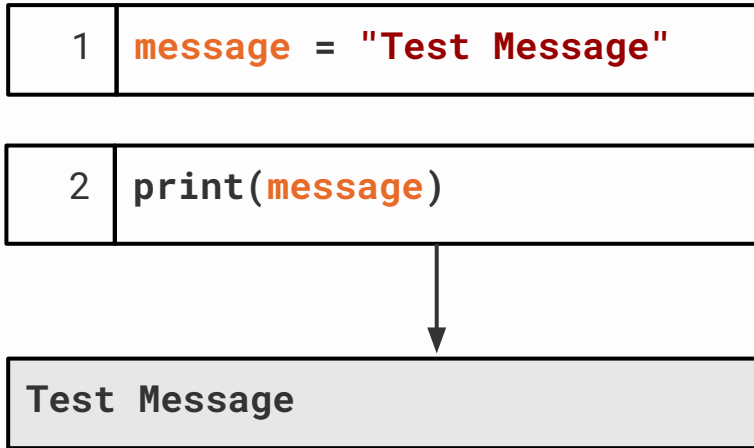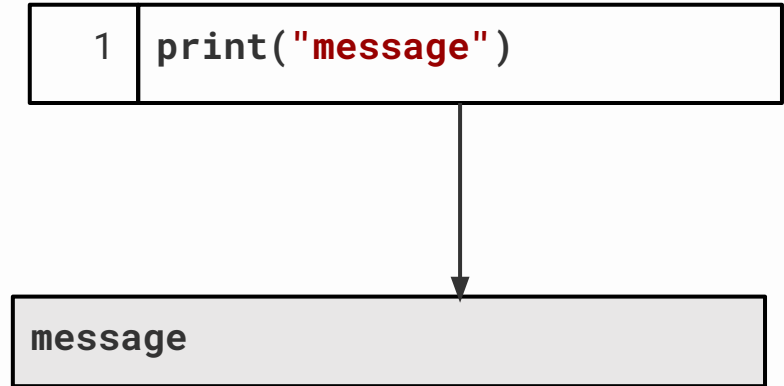
# Variable Naming

### Case Sensitive

Variables that differ even by one letter or casing are not the same

### No Special Chars

It only supports alphabetical letters or symbols and underscores

### Can do numbers

But it must not be the first part of the variable

# Quiz: Is this valid?

```
correct = "True"
```

```
years taken beforehand = 12
```

```
_hidden = "Please keep this a secret"
```

```
$var = 123
```

```
million_dollars = 1000000.00
```

```
何でもない = ""
```

# Variable as Nicknames

Variables are often used to represent data concisely

```
name = "José Protacio Rizal Mercado y Alonso Realonda"
```

```
print(name)
```

```
José Protacio Rizal Mercado y Alonso Realonda
```

# Change earlier code

Old Code:

```python
print("Hello! My name is your name")
print("I am learning Python")
```

New Code:

```python
name = "your name"
language = "Python"

print("Hello! My name is", name)
print("I am learning", language)
```

# Variable Change

Updating existing variables

# Variable Reassignment

Variables can be changed by using the same variable name

```
result = 9999
print(result)

result = 1
print(result)
```

```
9999
1
```

result
9999

result
1
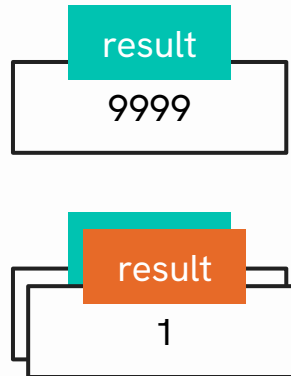
# Variable Evaluation

Python evaluates code top to bottom, right to left

← 

**age = 30**

**age = age + 1**

**age = 31**

```
age = 30
age = age + 1
```

| age |
|-----|
| 30 |

| age |
|-----|
| 30 + 1 |

| age |
|-----|
| 31 |

# Example 1: Level-up

Using a variable on the right side means using its current value *with some change*

```
1  level = 1
2  print("Level:", level)
3
4  # Player gains XP
5  level = level + 1
6  print("Leveled up! Level:", level)
```

# Example 2: Battery Level

Variables can be reassigned for as long as they are in scope (more on this later)

```python
1  battery = 100
2  print("Battery:", battery)
3
4  # Opened Chrome with 10 tabs
5  battery = battery - 40
6  print("After Chrome:", battery)
7
8  # Plugged in charger
9  battery = battery + 20
10 print("Charger inserted:", battery)
```

# Quick Exercise: Score System

```python
02_counter.py
```

```python
counter = 0
print("Counter:", counter)

# Point up: Add one to the counter
# TODO: Code here
print("Counter:", counter)

# Bonus: Multiply the score by 10
# TODO: Code here
print("Counter:", counter)

# Penalty: Decrease the score by 4
# TODO: Code here
print("Counter:", counter)
```

# Data Types

Built-in Information representation in Python

# Strings (str)

Strings represent text or a series of characters, enclosed in double or single quotes

```
empty_string_a = ''
```

```
empty_string_b = ""
```

```
quote = "I am a little teapot, short and spout."
```

What are other examples of strings?

# Integers (int)

Integers represent whole (no decimal), positive, or negative numbers

```
savings = 0
```

```
balance = -100
```

```
high_score = 111_222_333
```

What are other examples of integers?

# Floating-Point Numbers (float)

Floats represent real positive, or negative numbers with decimal points

```
temperature_celsius = 0.0
```

```
growth_rate = -0.56
```

```
avogadros_number = 6.022e+23
```

What are other examples of floats?

# Boolean (bool)

Booleans represent True or False

```
is_raining = True
```

```
exit_program = False
```

What are other examples of booleans?

# None (None)

None represent null or empty values

```
response = None
```

# Quick Exercise: Wishlist

```python
03_wishlist.py
```

```python
1  # Fill in the variables based on the item you want to buy
2  name = What is the name of the item?
3  price = How much is the item?
4  organic = Is it organic?
5
6  # Then, print each information one line at a time
7  print(name)
8  print(price)
9  print(organic)
```

# Input Function

Using data given by the user of the code

# Input Function

The **input** function gets data given in the console. The given data can then be stored in a variable. Note: The input will always return a str.

```
1  user_input = input()
2  print(user_input)
```

```
1  user_input = input("Enter input: ")
2  print(user_input)
```

| User | → | user_input |
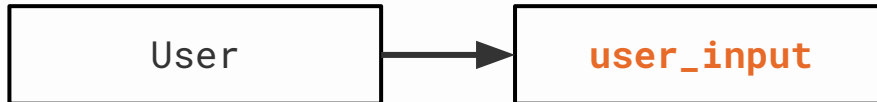
# Quick Exercise: Expense Tracker

```
04_expense_tracker.py
```

```python
1  # Ask the user for three values
2  expense_1 = Let the user enter a number
3  expense_2 = Let the user enter a number
4  expense_3 = Let the user enter a number
5
6  # Then, print each information one line at a time
7  print(expense_1)
8  print(expense_2)
9  print(expense_3)
```

# Operations

Applying transformations to data

# Number Operations

| Symbol | Operation | Example | Result |
|--------|-----------|---------|--------|
| + | Addition | `result = 11 + 2` | 13 |
| - | Subtraction | `result = 11 - 2` | 9 |
| * | Multiplication | `result = 11 * 2` | 22 |
| / | Division | `result = 11 / 2` | 5.5 |

# Quick Exercise: Expense Tracker (v2)

**04_expense_tracker.py**

```python
# Ask the user for three values
expense_1 = Let the user enter a number
expense_2 = Let the user enter a number
expense_3 = Let the user enter a number

# Then, print each information one line at a time
print(expense_1)
print(expense_2)
print(expense_3)

total = Calculate the sum of the numbers
print(total)
```

# Strings and Numbers

Be careful not to confuse integers and strings that look like integers

```
1  number = "123"
2  number = number + 1
```

≠

```
1  number = 123
2  number = number + 1
```

# Integer Type Conversion

You can convert most basic data types to an integer with `int()` function.

```python
number1 = input("Enter number: ")
number2 = input("Enter number: ")
total = number1 + number2
print(total)
```

```python
number1 = int(input("Enter number: "))
number2 = int(input("Enter number: "))
total = number1 + number2
print(total)
```

# Integer Type Conversion

| Original Data Type | Result |
|---|---|
| **Float** | Drops all decimal places |
| **Boolean** | True $\rightarrow$ 1, False $\rightarrow$ 0 |
| **String** | Converts to integer. If invalid, **raises an error** |
| **None** | **Raises an error** |

# Float Type Conversion

| Original Data Type | Result |
|---|---|
| Integer | Adds .0 decimal place |
| Boolean | True → 1.0, False → 0.0 |
| String | Converts to float. If invalid, **raises an error** |
| None | **Raises an error** |

# Quick Exercise: Expense Tracker (v2)

**04_expense_tracker.py**

```python
# Ask the user for three values
expense_1 = Let the user enter a number
expense_2 = Let the user enter a number
expense_3 = Let the user enter a number

# Then, print each information one line at a time
print(expense_1)
print(expense_2)
print(expense_3)

total = Calculate the sum of the numbers
print(total)
```

# Operations+

Applying more complex transformations to data

# Floor Division

The floor division operator divides the number on the left by the right and rounds down

```python
1  division = 11 / 2
2  print(division)
3
4  floor_division = 11 // 2
5  print(floor_division)
```

```
5.5
5
```

# Exponent/Power Operator

The exponent operator multiplies the number on the left by itself multiple times

```
1  result = 2 ** 4
2  print(result)
3
4  manual_result = 2 * 2 * 2 * 2
5  print(manual_result)
```

```
16
16
```

# Modulo/Remainder Operator

The modulo operator returns the remainder if the left side was divided by the right side

```
1  result = 11 % 3
2  print(result)
```

2

$$11 \div 3 \qquad = \qquad 3 \text{ remainder } 2$$

Step 1:  $3 \times 3 = 9$  (3 fits into 11 three times)

Step 2:  $11 - 9 = 2$  (remainder is 2)

# Order of Operations

Given the following operation:

$$3 + 5 \times 2 - \frac{8}{4}$$

This can be translated in Python with the following expression:

```python
result = 3 + 5 * 2 - 8 / 4
print(result)
```

```python
result = 3 + (5 * 2) - (8 / 4)
print(result)
```

# String Operations

Two common operations for strings

# String Concatenation (Addition)

Multiple strings can be combined using the addition operator

```
1  print("Hello" + " " + "Hello")
```

```
Hello World
```

# String Repetition (Multiplication)

A string can be repeated multiple times using the multiplication operator.

```
1  print("ice " * 3)
```

```
ice ice ice
```

# Challenge: Ice Ice Ice Baby

05_ice_ice_ice_baby.py

```python
ice = "Ice"
baby = "Baby"

# Print "Ice Ice Ice Baby" using + and *
print()
```

# Updates

Shortcut for reassignments

# Update Shortcut

All operations where the variable is changed by a copy of it can be simplified

| Original Statement | Shortcut |
|---|---|
| `result = result + 5` | `result += 5` |
| `result = result * 10` | `result *= 10` |
| `message = message + "World"` | `message += "World"` |

# Example 1:  Level-up (Updated)

Using a variable on the right side means using its current value *with some change*

```
1  level = 1
2  print("Level:", level)
3
4  # Player gains XP
5  # Previously: level = level + 1
6  level += 1
7  print("Leveled up! Level:", level)
```

# Example 2: Battery Level (Updated)

Variables can be reassigned for as long as they can be accessed (again, more on this later)

```python
battery = 100
print("Battery:", battery)

# Opened Chrome with 10 tabs
# Previously: battery = battery - 40
battery -= 40
print("After Chrome:", battery)

# Plugged in charger
# Previously: battery = battery + 20
battery += 20
print("Charger inserted:", battery)
```

# Quick Exercise: Score System (v2)

**02_counter.py**

```python
counter = 0
print("Counter:", counter)

# Point up: Add one to the counter
# Change your code in this line
print("Counter:", counter)

# Bonus: Multiply the score by 10
# Change your code in this line
print("Counter:", counter)

# Penalty: Decrease the score 4
# Change your code in this line
print("Counter:", counter)
```

# String Formats

Combine strings and variables conveniently

# String Placeholder

```python
# Message Template
message = "Hello {}! Nice to meet you!"
print(message)

# Use Template
formatted_message = message.format("Juan")
print(formatted_message)
```

```
Hello {}! Nice to meet you!
Hello Juan! Nice to meet you!
```

# String Placeholder (Repeated Use)

```python
# Message Template
message = "Hello {}! Nice to meet you!"
print(message)

# Use Template
formatted_message = message.format("Juan")
print(formatted_message)

# Use Template (again)
new_message = message.format("Jesse")
print(new_message)
```

```
Hello {}! Nice to meet you!
Hello Juan! Nice to meet you!
Hello Jesse! Nice to meet you!
```

# Multiple String Formatting

The format method supports multiple inputs as well as needed.

```python
message = "Hello {}. Your nickname is {}"

name = input("Enter name: ")
nickname = input("Enter nickname: ")

formatted_message = message.format(name, nickname)
print(formatted_message)
```

# Multiple String Formatting (Named)

Placeholders can be given a variable name to make assignment easier

```python
message = "Hello {first}. Your nickname is {second}"

name = input("Enter name: ")
nickname = input("Enter nickname: ")

formatted_message = message.format(first=name, second=nickname)
print(formatted_message)
```

# Quick Exercise: Price Post

```python
06_price_post.py
```

```python
1  # Price notification template
2  price_notification = "The price of {} is ${}."
3
4  # Post: Latte ($3.5)
5  print(price_notification)
6
7  # Post: Espresso ($2.75)
8  print(price_notification)
9
10 # Post: Cappuccino ($4.0)
11 print(price_notification)
```

# String Formatting (Modern)

This is the old format that is still used to this day

```
1  name = input("Enter your name: ")
2  print("Hello {} Nice to meet you!".format(name))
```

For short strings, the modern f-string format is used

```
1  name = input("Enter your name: ")
2  print(f"Hello {name} Nice to meet you!")
```

# Quick Exercise: Expense Tracker (v3)

```
04_expense_tracker.py
```

```
1   # Ask the user for three values
2   expense_1 = Let the user enter a number
3   expense_2 = Let the user enter a number
4   expense_3 = Let the user enter a number
5
6   # Then, print each information one line at a time
7   print(expense_1)
8   print(expense_2)
9   print(expense_3)
10
11  total = expense_1 + expense_2 + expense_3
12  print(total)
13
14  print(expense_1, "+", expense_2, "+", expense_3, "=", total)
```

# Sales Tracker

Quick practice of all the concepts discussed so far

# Quick Exercise: Sales Tracker

**07_sales_tracker.py**

```python
# Ask the cost and pax or count for three separate items
item_cost_1 = Let the user enter a number
item_count_1 = Let the user enter a number

item_cost_2 = Let the user enter a number
item_count_2 = Let the user enter a number

item_cost_3 = Let the user enter a number
item_count_3 = Let the user enter a number

# Calculate the total
total = 0
print(total)
```

**03**

# Control Flow

Providing logic to data processing

# Relations

Checking if two values are related to each other

EXAM FORM

Page 1

# What are the possible results?

**number_1 > number_2**

True          False

# The result can be stored

**result** = **number_1** > **number_2**

**True**          **False**
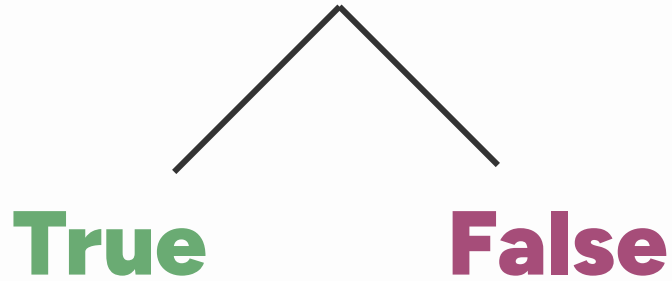
# Relational Operator

All of the basic data types (except None) support relational operator (returns a `bool`)

| Symbol | Operation | Example | Value |
|--------|-----------|---------|-------|
| < | Less Than | 11 < 2 | **False** |
| <= | Less than or Equal | 11 <= 2 | **False** |
| > | Greater Than | 11 > 2 | **True** |
| >= | Greater Than or Equal | 11 >= 2 | **True** |

Remember: PACMAN First

# Quick Exercise: Height Requirement

08_height_requirement.py

```python
1  minimum_height = 138
2
3  # Ask the user for the following inputs
4  user_height = User height (in cm)
5
6  # Notify user if they can enter the ride
7  can_enter_ride = None
8  print("Can enter the ride:", can_enter_ride)
```

# Chained Relational Operator

Similar to the mathematical notation, relational operators can be chained to ask for ranges

```python
x = int(input("Enter number: "))

print("Exclusive Range")
print(3 < x < 20)

print("Equal or Greater than 3 and Less than 20")
print(3 <= x < 20)

print("Greater than 3 and Less than or Equal to 20")
print(3 < x <= 20)

print("Inclusive Range")
print(3 <= x <= 20)
```

# Quick Exercise: Valid Score

**09_valid_score.py**

```python
# Range minimum and maximum bounds
min_number = 0
max_number = 100

# Enter user input
number = Enter number

# Notify user if the number is a valid score
valid_score = None
print("Valid score:", valid_score)
```

# Value (In)equality

The most common relation operator is the equal and not equal operators

| Symbol | Operation | Integer Example | String Example |
|--------|-----------|-----------------|----------------|
| `==`   | Equal     | `11 == 2`       | `"Hello" == "World"` |
| `!=`   | Not Equal | `11 != 2`       | `"Hello" != "World"` |

# Example: Perfect Score Check

Similar to the mathematical notation, relational operators can be chained to ask for ranges

```
1  score = int(input("Enter score: "))
2  perfect_score = 100
3
4  has_perfect_score = score == perfect_score
5
6  print("You got a perfect score:", has_perfect_score)
```

# Quick Exercise: Login System

```python
10_login_system.py
1  # Expected password (you can change the value)
2  correct_password = "pass"
3
4  # Enter user password
5  password_input = input("Please provide password: ")
6
7  # Notify user if password is valid
8  correct_password_given = None
9  print("Access Granted", correct_password_given)
```

# Conditionals

Control when code executes

# If Statement - True

```python
1  login_input = input("Login: ")
2
3  if login_input == "Yes":
4      print("Welcome")
5      print("Back")
6  print("End")
```

Input: "Yes"

```
Welcome
Back
End
```

logged_in
==
"Yes"

True

Welcome

Back

End
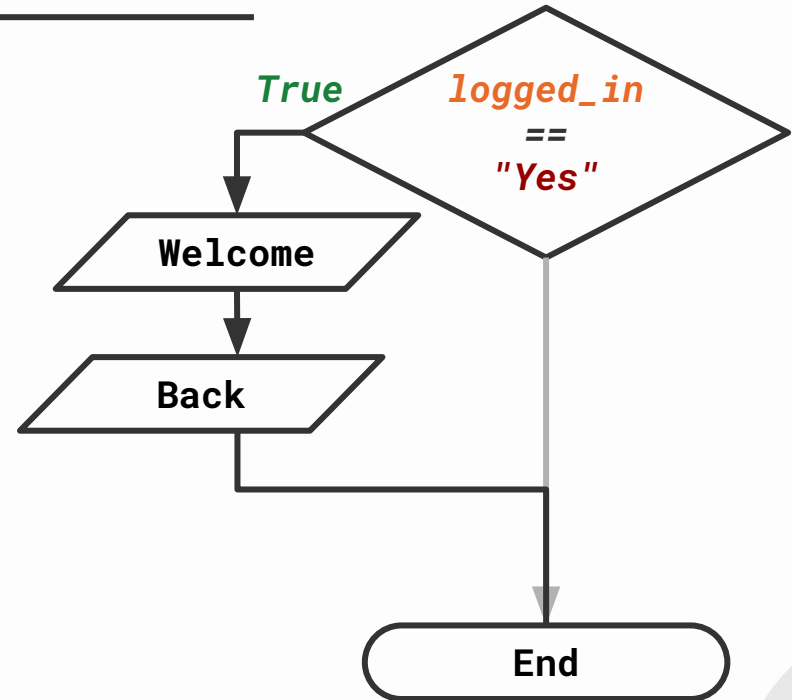
# If Statement - False

```
1  login_input = input("Login: ")
2
3  if login_input == "Yes":
4      print("Welcome")
5      print("Back")
6  print("End")
```

Input: *"No"*

```
Welcome
Back
End
```

logged_in
==
"Yes"

False

Welcome

Back

End
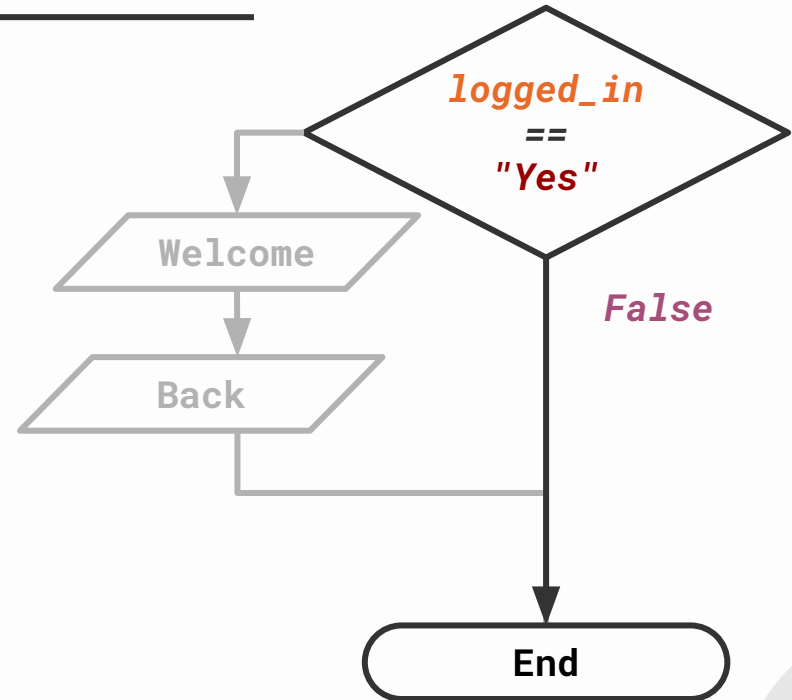
# Quick Exercise: Login System (v2)

**10_login_system.py**

```python
# Expected password (you can change the value)
correct_password = "pass"

# Enter password
password_input = input("Please provide password: ")

# Notify user if password is valid
correct_password_given = None
print("Access Granted")
```

# Elif Statement

The else-if or elif statements allow you to run parts of the code when the first condition is **False** but there are other possible options
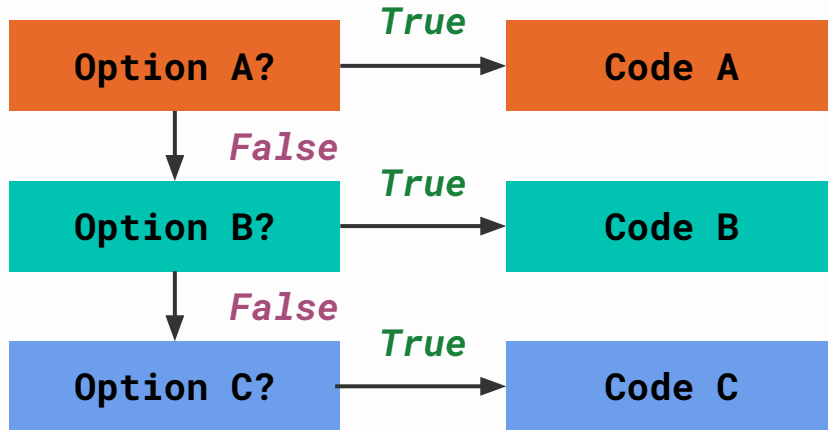


```python
if condition_1:
    """Code A"""

elif condition_2:
    """Code B"""

elif condition_3:
    """Code C"""
```

# Elif Statement Example 01

```
1   you_said = input("You said: ")
2
3   if you_said == "Wish":
4       print("107.5")
5   elif you_said == "Hello":
6       print("...it's me")
7   elif you_said == "Jopay":
8       print("...kamusta ka na")
9   elif you_said == "Black Pink":
10      print("...in your area")
```

*...kamusta ka na*

you_said

107.5

it's me

Kamusta ka na

In you area

End
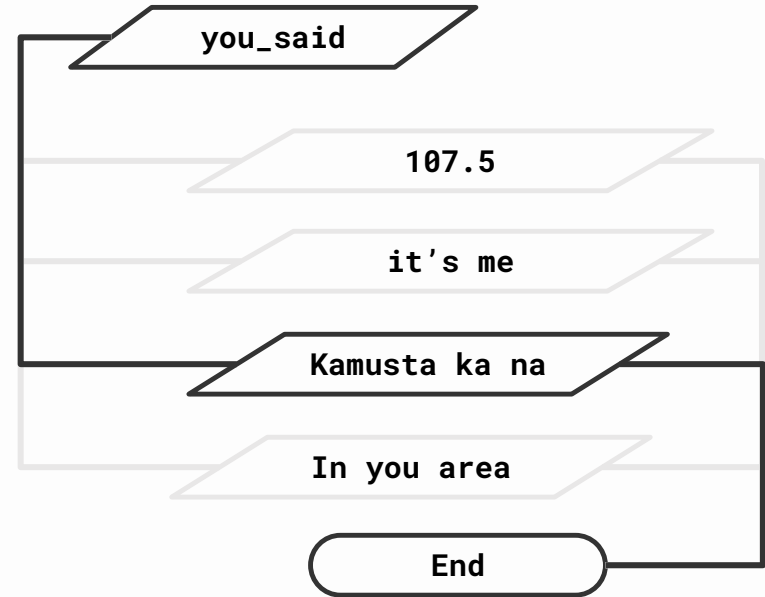
# Elif Statement Example 02

```python
battery = int(input("Battery percentage: "))

if battery >= 80:
    print("Full Battery")
elif battery >= 40:
    print("Good Battery")
elif battery >= 15:
    print("Low Battery")
elif battery > 0:
    print("Critically Low Battery")
```

# Quick Exercise: Greetings

**11_traffic_lights.py**

```python
# Ask the user input for a color
color_input = input("Please enter a color: ")

# Print the following depending on the color input
# "green"   -> print "Go"
# "yellow"  -> print "Wait..."
# "red"     -> print "Stop"
```

# Else Statement (Last Resort)

The else statement runs a piece of code when every condition fails
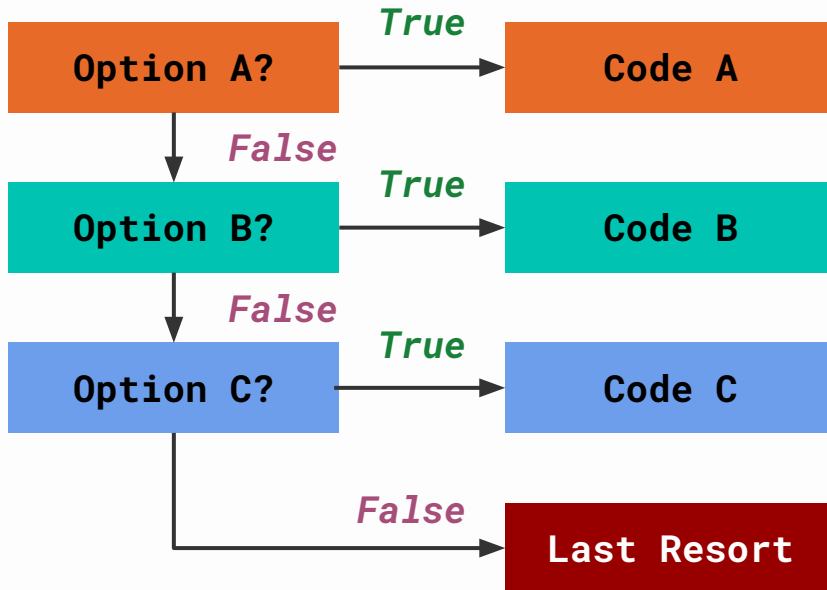


```python
if condition_1:
    """Code A"""

elif condition_2:
    """Code B"""

elif condition_3:
    """Code C"""

else:
    """Last Resort"""
```

# Else Statement

```
1   you_said = input("You said: ")
2
3   if you_said == "Wish":
4       print("107.5")
5   elif you_said == "Hello":
6       print("...it's me")
7   elif you_said == "Jopay":
8       print("...kamusta ka na")
9   elif you_said == "Black Pink":
10      print("...in your area")
11  else:
12      print("I don't know")
```

you_said

107.5

it's me

Kamusta ka na

In you area

I don't know

End
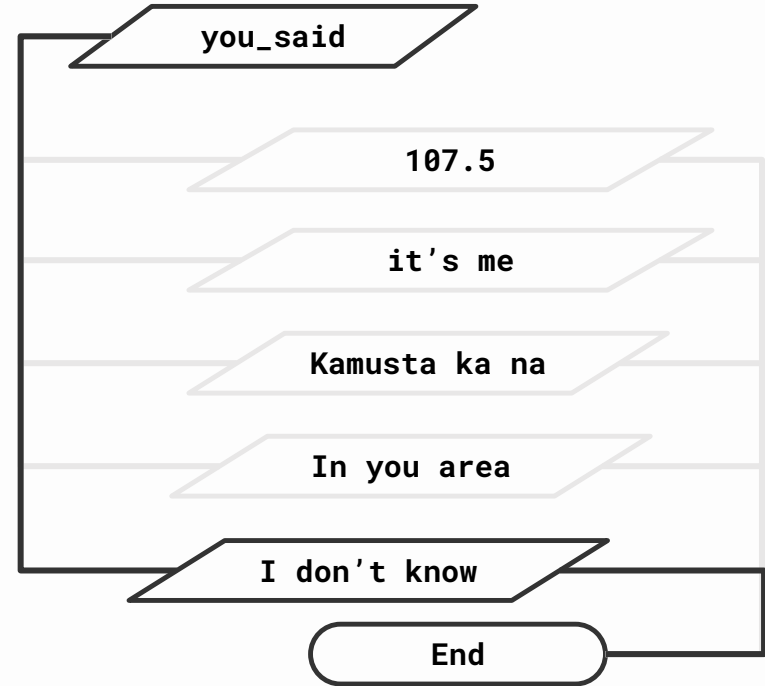
# Quick Exercise: Traffic Lights (version 2)

**11_traffic_lights.py**

```python
# Ask the user input for a color
color_input = input("Please enter a color: ")

# Print the following depending on the color input
# if green
#    -> "Go"
# elif yellow
#    -> "Wait..."
# elif red
#    -> "Stop"
# else
#    -> "Malfunction"
```

# Multiple If's versus If-Elif's

If-elif statements ensure only one option runs. That's not the case for multiple if statements

```python
1  grade = 85
2
3  if grade >= 90:
4      print("A")
5  if grade >= 80:
6      print("B")
7  if grade >= 70:
8      print("C")
```

```
B
C
```

```python
1  grade = 85
2
3  if grade >= 90:
4      print("A")
5  elif grade >= 80:
6      print("B")
7  elif grade >= 70:
8      print("C")
```

```
B
```

# If-Else Condition

```python
age = int(input("Enter age: "))
if age >= 18:
    print("Old enough to watch movie")
else:
    print("Too young to watch movie")
```

```python
balance = 150
price = 200

if balance >= price:
    print("Payment successful")
else:
    print("Insufficient funds")
```

# Quick Exercise: Login System (v3)

**10_login_system.py**

```python
# Expected password (you can change the value)
correct_password = "pass"

# Enter password
password_input = input("Please provide password: ")

# Notify user if password is valid or invalid
correct_password_given = None
print("Access Granted")
print("Access Denied")
```

# Logical Operators

Simplifying conditionals

# And Operator

Use the **and** operator to restrict conditions

```python
applied = True
has_skill = True
has_experience = True

if applied and has_skill and has_experience:
    print("You're hired")
```

# And Operator Example

You can use the `and` operator to make the condition more strict

```python
money = float(input("Enter money: "))
stock = int(input("Enter stock: "))

if money >= 100 and stock > 0:
    print("You can buy the item!")
else:
    print("You can't buy the item")
```

# Quick Exercise: Login System (v4)

```python
10_login_system.py
1  # Expected username and password (you can change the value)
2  correct_username = "user"
3  correct_password = "pass"
4
5  # Enter username and password
6  username_input = input("Please provide username: ")
7  password_input = input("Please provide password: ")
8
9  # Notify user if credentials are valid or invalid
10 correct_credentials = None
11 print("Access Granted")
12 print("Access Denied")
```

# Or Operator

Use the `or` operator to add alternative conditions

```python
1  raining = False
2  cold = False
3  trendy = True
4
5  if raining or cold or trendy:
6      print("Wear a jacket")
```

# Or Operator Example

Use the `or` operator to add alternative conditions

```python
1  response = input("Continue? ")
2  if response == "yes" or response == "YES":
3      print("We will continue!")
```
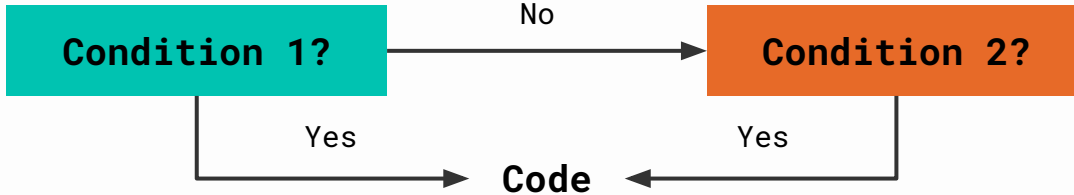
# Quick Exercise: Login System (v5)

```
10_login_system.py
```

```python
1  # Expected username and password (you can change the value)
2  correct_username = "user"
3  correct_password = "pass"
4  admin_username = "admin"
5  admin_password = "admin"
6
7  # Enter username and password
8  username_input = input("Please provide username: ")
9  password_input = input("Please provide password: ")
10
11 # Notify user if credentials are valid or invalid
12 correct_credentials = None
13 print("Access Granted")
14 print("Access Denied")
```

# Not Operator

A boolean value or statement can be reversed or negated using the not operator

```python
1  print(not True)
```

```python
2  print(not False)
```

```python
3  correct_credentials = False
4  if not correct_credentials:
5      print("Access Denied")
```

# For Loops

Controlled repetitions

# Defining a List

```
items = ["milk","egg","ice"]
print( items )
```

# Implement: Bookmarks

```python
# Define a list of your favorite websites
websites = ["facebook.com", "youtube.com"]

# Print the entire list of websites
print(websites)
```

**For Loop**

```
items = ["milk","egg","ice"]
for item in items:
        print( item )
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

| "milk" | "egg" | "ice" |
|--------|-------|-------|

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

*item*="milk"

| "milk" | "egg" | "ice" |
|--------|-------|-------|

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

milk

item="milk"

| "milk" | "egg" | "ice" |

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

*milk*

*item*="egg"

| "milk" | "egg" | "ice" |
|--------|-------|-------|

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

```
milk
egg
```

*item*="egg"

| "milk" | "egg" | "ice" |
|--------|-------|-------|

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

```
milk
egg
```

*item*="ice"

| "milk" | "egg" | "ice" |
|--------|-------|-------|

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop

A for loop goes through the items or elements of a list one at a time by assigning to a variable

```
1  items = ["milk","egg","ice"]
2  for item in items:
3      print(item)
```

```
milk
egg
ice
```

*item*="ice"

| "milk" | "egg" | "ice" |
|--------|-------|-------|

```
1  item = "milk"
2  print(item)
3
4  item = "egg"
5  print(item)
6
7  item = "ice"
8  print(item)
```

# For Loop Example 01: Prints

For prints are often used to print values one at a time

```
1  notifications = ["Battery low", "New message", "New Update"]
2
3  for notification in notifications:
4      print("Alert:", notification)
```

```
Alert: Battery low
Alert: New message
Alert: New Update
```

# For Loop Example 02: Aggregation

A common task in for loops is combining all of the items into one value

```python
expenses = [1500, 2300, 1200]
total = 0

for amount in expenses:
    total += amount

print("Total expenses:", total)
```

```
Total expenses: 5000
```
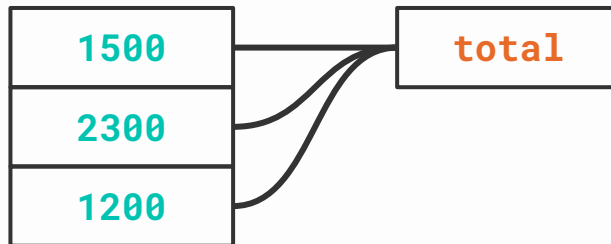
# For Loop Example 03: Counting

Finally, another common task besides aggregation is counting

```
1  grades = [85, 72, 49, 90, 66, 59]
2  passing = 0
3
4  for grade in grades:
5      if grade >= 60:
6          passing += 1
7
8  print("Passing:", passing)
```

*Passing: 4*

| 85 |
| 72 |
| 49 |
| 90 |
| 66 |
| 59 |

total
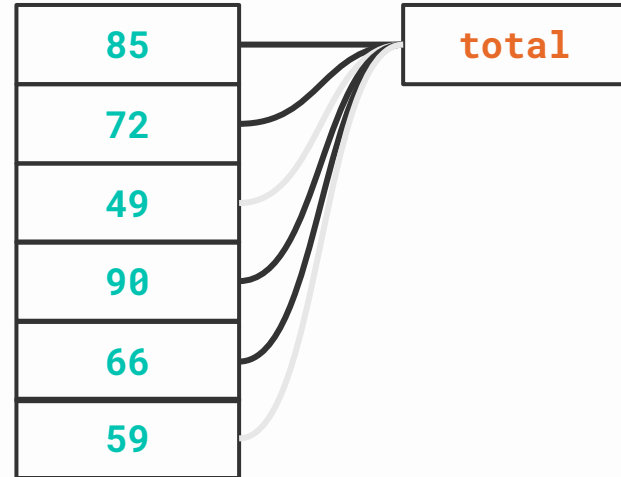
# Quick Exercise: Bookmarks (v2)

**12_bookmarks.py**

```python
# Define a list of your favorite websites (add or change below)
websites = ["facebook.com", "youtube.com"]

# Print the entire list of websites (one at a time)
print(websites)
```

# Fixed Repetition

Using a *range(n)* function instead of a list makes the code repeat that many times

```
1  for item in range(3):
2      print("This will be repeated")
```

```
This will be repeated
This will be repeated
This will be repeated
```

# Quick Exercise: Repetition

13_repetition.py

```python
# Long Message
message = "This is a very long message that's hard to type"

# Print the message eleven times
print(message)
```

# For Range Loop

The *range(n)* function actually generates a list from *0* to *n-1*

```
1  for item in range(3):
2      print(item)
```

```
0
1
2
```

```
1  numbers = [0, 1, 2]
2  for item in numbers:
3      print(item)
```

# Quick Exercise: Counting

**14_counting.py**

```python
# Ask the user for a number
end = int(input("Enter number: "))

# Print the numbers 0 to end
print()
```

# Range() with different start

The *range(start,end)* is a variation of *range(n)* function that generates a list from *start* to *end-1*

```
1  for item in range(1, 6):
2      print(item)
```

```
1
2
3
4
5
```

# Quick Exercise: Counting (v2)

```
14_counting.py
1  # Ask the user for a starting and ending number
2  start = int(input("Enter start: "))
3  end = int(input("Enter end: "))
4
5  # Print the numbers start to end
6  print()
```

# Range() with different step

The *range*(*start*, *end*, *step*) is a variation of *range*(*n*) function that generates a list from *start* to *end-1* and skips count by *step*

```
1  for item in range(2, 11, 2):
2      print(item)
```

```
2
4
6
8
10
```

# Quick Exercise: Tens

```
15_tens.py
1  # Print the following pattern up to 100
2  # 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
3
4  print()
```

# While Loops

Dynamic repetitions

# Dynamic Repetition

Some repetitions cannot be fixed or determined

```
1  correct_password = "pass"
2
3  password = input("Enter password: ")
```

# Dynamic Repetition

Some repetitions cannot be fixed or determined

```python
1  correct_password = "pass"
2
3  password = input("Enter password: ")
4  if password != correct_password:
5      password = input("Enter password: ")
```

# Dynamic Repetition

Some repetitions cannot be fixed or determined

```
1  correct_password = "pass"
2
3  password = input("Enter password: ")
4  if password != correct_password:
5      password = input("Enter password: ")
6
7      if password != correct_password:
8          password = input("Enter password: ")
```

# Dynamic Repetition

Some repetitions cannot be fixed or determined

```
1  correct_password = "pass"
2
3  password = input("Enter password: ")
4  if password != correct_password:
5      password = input("Enter password: ")
6
7      if password != correct_password:
8          password = input("Enter password: ")
9
10         if password != correct_password:
11             password = input("Enter password: ")
```

# Repeat until the user gets it right

```
password = input()
while password != "pass":
    password = input()
```

# While Loop Example

This structure is commonly used to repeat certain tasks until user says otherwise

```python
running = True
while running:
    command = input("Provide command: ")
    if command == "command 1":
        print("command 1 done")
    elif command == "command 2":
        print("command 2 done")
    elif command == "command 3":
        print("command 3 done")
    elif command == "exit":
        running = False
```

# Quick Exercise: Running Balance

**16_running_balance.py**

```python
total = 0
running = True
while running:
    command = input("Provide command: ")

    if command == "add":
        # Ask for number, add to total, and print
    if command == "sub":
        # Ask for number, subtract to total, and print
    elif command == "exit":
        running = False
```

# Loop Control

Exit the common mold

# Break Keyword

The **break** keyword immediately stops the loop

```
1  for item in range(100):
2      print(item)
3      if item == 3:
4          break
```

break

| item = 0 | item = 1 | item = 2 | item = 3 | item = 4 |
|----------|----------|----------|----------|----------|
| print(item) | print(item) | print(item) | print(item) | print(item) |
| break | break | break | break | break |
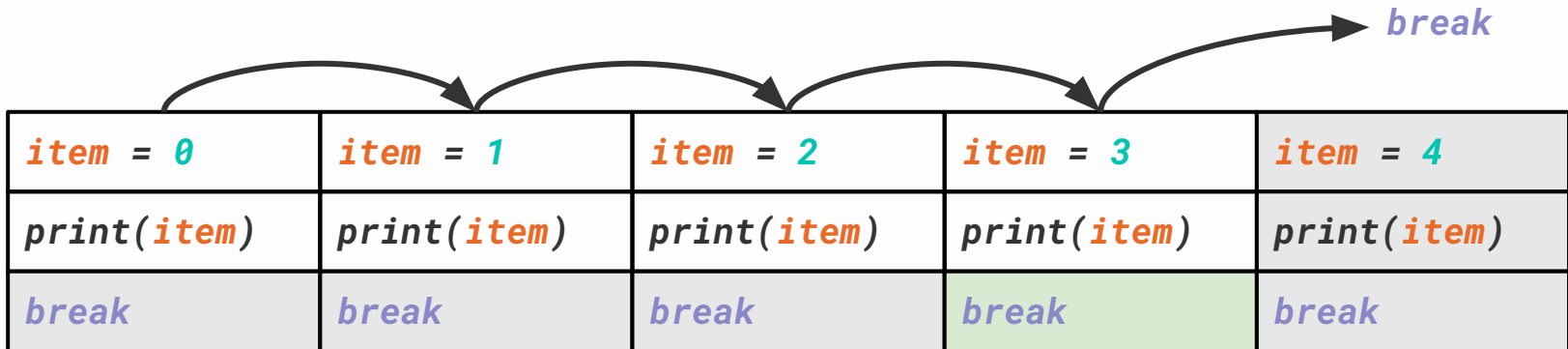
# While Loop: Password Attempt

```python
max_attempt = 3
correct_password = "pass"

for attempt in range(max_attempt):
    password = input("Enter password: ")
    if password == correct_password:
        print("Access granted")
        break
```

# Quick Exercise: Search

```
17_search.py
1   items = ["rice", "noodles", "toyo", "spam", "coffee"]
2   item_to_find = "spam"
3
4   for item in items:
5       """If item equals the item_to_find, print and exit loop"""
6
7
```

# Continue Keyword

The **continue** keyword skips the succeeding code

```python
1  for item in range(100):
2      if item == 3:
3          continue
4      print(item)
```

| continue | continue | continue | continue | continue |
|---|---|---|---|---|
| print(item) | item = 1 | item = 2 | item = 3 | item = 4 |
| item = 0 | print(item) | print(item) | print(item) | print(item) |
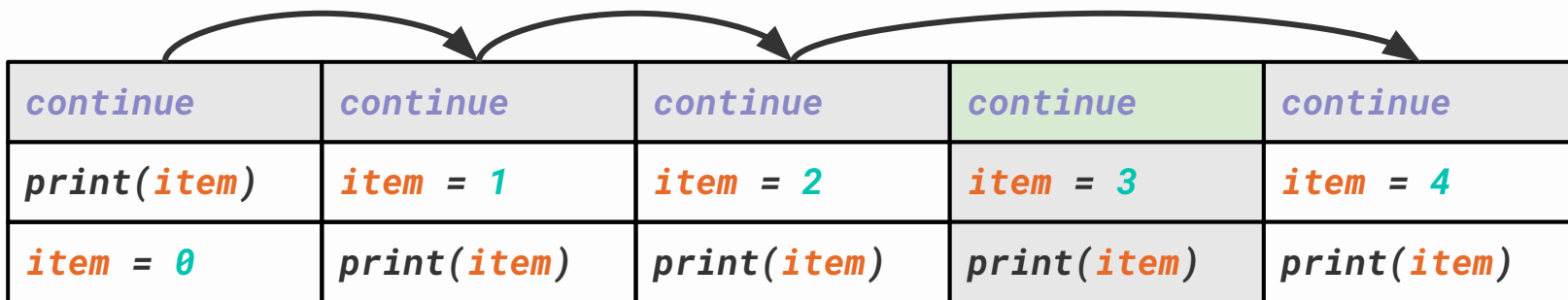
# Quick Exercise: Skip Range

18_skip_range.py

```python
for item in range(100):
    # Change code to skip printing numbers 20 to 80.
    print(item)
```

**H2**

# Sales Tracker v2

Make the previous version more dynamic!

# Sales Tracker (v2)

```
07_sales_tracker.py
```

```
 1  input_count = Ask the user how many items will be calculated
 2
 3  total = 0
 4
 5  # Use a for loop to ask for more than one cost and count
 6  item_cost = Let the user enter a number
 7  item_count = Let the user enter a number
 8  item_total = item_cost * item_count
 9
10  print(total)
```

# 04

# Functions

First step to code organization

# Sum Calculator

```
1  numbers = [1, 3, 5, 7, 2, 4, 6]
2  total = 0
3  for number in numbers:
4      total += number
5
6  print(total)
```

```
7  new_numbers = [9, 3, 0, 1, 2, 7]
```

# Sum Calculator

```
1  numbers = [1, 3, 5, 7, 2, 4, 6]
2  total = 0
3  for number in numbers:
4      total += number
5
6  print(total)
```

```
7   new_numbers = [9, 3, 0, 1, 2, 7]
8   total_2 = 0
9   for number in new_numbers:
10      total_2 += number
11
12  print(total_2)
```

**What if I need to calculate another list?**

# Sum Calculator

```
1  numbers = [1, 3, 5, 7, 2, 4, 6]
2  total = sum(numbers)
3  print(total)
```

```
4  new_numbers = [9, 3, 0, 1, 2, 7]
5  total_2 = sum(new_numbers)
6  print(total_2)
```

**No need to copy paste code**

# Function Copy-Pasting

```
1  def extra():
2      print("Extra Line 1")
3      print("Extra Line 2")

4
5  print("First Line")
6  extra()
7  print("Second Line")
```

```
1  print("First Line")
2
3  print("Extra Line 1")
4  print("Extra Line 2")
5
6  print("Second Line")
```

```
First Line
Extra Line 1
Extra Line 2
Second Line
```

# Simple Function Declaration

```python
def function_name():
    """processes here"""
```

```python
1  def greet():
2      print("Hello, good day to you!")
```

```python
3  greet()
```

# Quick Exercise: Line Generator

**19_line_generator.py**

```python
"""
Create a function line_generator that prints the following:
    Line 1
    Line 2
    Line 3
"""

# Use the function once
line_generator()
```

```python
def function_name():
    """processes here"""
```

# Simple Input Declaration

```
def function_name(variable_name):
    """processes here"""
```

```
1  def greet(username):
2      print(f"Hello {username}, good day to you!")
```

```
3  greet("Joseph")
```

# Quick Exercise: Line Generator (version 2)

```python
"""
Create a function line_generator that has a parameter number
and prints the following:
    Line 1
    Line 2
    ...
    Line number
"""

# Use the function once
line_generator(4)
```

# Multiple Input Declaration

```python
def function_name(variable_name_1, variable_name_2):
    """processes here"""
```

```python
1  def greet(username, message):
2      print(f"Hello {username}, {message}")
```

```python
3  greet("Joseph", "Nice to meet you!")
```

# Quick Exercise: Product

```
20_product.py
1  def product():
2      """Takes three inputs and print the product"""
3
4  product(1, 1, 1) # 1
5  product(1, 2, 3) # 6
6  product(2, 5, 10)# 100
```

# Optional Parameter

```python
def function_name(variable_name_1, variable_name_2=default):
    """processes here"""
```

```python
1  def greet(username, message="Nice to meet you!"):
2      print(f"Hello {username}, {message}")
```

```python
3  greet("Joseph")
```

# Optional Parameter (Overriding)

```python
def function_name(variable_name_1, variable_name_2=default):
    """processes here"""
```

```python
1  def greet(username, message="Nice to meet you!"):
2      print(f"Hello {username}, {message}")
```

```python
3  greet("Joseph", "Hajimemashite!")
```

# Quick Exercise: Product (v2)

```
20_product.py
```

```python
def product():
    """Takes three inputs (or two inputs) and print the product"""

product(1, 1, 1) # 1
product(1, 2, 3) # 6
product(2, 5, 10)# 100
product(3, 3)    # 9
product(2, 5)    # 10
```

# Function Returns

Simplifying calculations and data handling

# Return Value

```python
def function_name(...):
    """processes here"""
    return output
```

```python
1  def add(num1, num2):
2      result = num1 + num2
3
4
5  add(1, 2)
6  print()
```

```python
1  def add(num1, num2):
2      result = num1 + num2
3      return result
4
5  final = add(1, 2)
6  print(final)
```

# Return Value

```
def function_name(...):
    """processes here"""
    return output
```

```
1  def add(num1, num2):
2      result = num1 + num2
3
4
5  final = result * 5
6  print(final)
```

```
1  def add(num1, num2):
2      result = num1 + num2
3      return result
4
5  final = add(1, 2) * 5
6  print(add_result)
```

# Return versus Print

The return keyword does not print the value in the console

```
1  def add(num1, num2):
2      result = num1 + num2
3      return result
4
5  add(1, 2)
```

```
1  def add(num1, num2):
2      result = num1 + num2
3      print(result)
4
5  add(1, 2)
```

*3*

# Return versus Print

The return keyword allows you to store the value in a variable instead

```python
1  def add(num1, num2):
2      result = num1 + num2
3      return result
4
5  add_result = add(1, 2)
6  print(add_result)
```

*3*

```python
1  def add(num1, num2):
2      result = num1 + num2
3      print(result)
4
5  add_result = add(1, 2)
6  print(add_result)
```

*None*

# Return Function Example 01

Return functions are used to provide context to a calculation

```
1  def to_fahrenheit(celsius):
2      return celsius * 9/5 + 32
3
4  print(to_fahrenheit(30))
```

# Return Function Example 02

Functions are also used to augment strings

```python
1  def happy(string):
2      return string + " :D"
3
4  message = "Hello World"
5  happy_message = happy(message)
6  print(happy_message)
```

# Return is Final!

When you return in a function it skips everything else after it!

```
1  def add(num1, num2):
2      total = num1 + num2
3      return total
4      print("Code finished")        ←———  skipped
```

```
5  result = add(3, 4)
6  print(result)
```

# Quick Exercise: Product (v3)

```
20_product.py
```

```python
def product():
    """Takes three inputs (or two inputs) and return"""

product(1, 1, 1) # 1
product(1, 2, 3) # 6
product(2, 5, 10)# 100
product(3, 3)    # 9
product(2, 5)    # 10
```

# Function Scope

Determining variable lifetime

# Functions can read outside

Function can detect and print variables outside of it

```python
outer_variable = 10

def function():
    print("Inner", outer_variable)

function()
print("Outer", outer_variable)
```

**Namespace**

| outer_variable | 10 |
| --- | --- |

```
Inner: 10
Outer: 10
```

# Functions can't write outside

Function can't edit variables outside normally

```python
outer_variable = 10

def function():
    outer_variable = 999
    print("Inner", outer_variable)

function()
print("Outer", outer_variable)
```

```
Inner: 999
Outer: 10
```

Namespace

| outer_variable | 10 |
|---|---|
| function<br>.outer_variable | 999 |

# Functions update using return

Function can interact outside using a return statement

```python
variable = 10

def pow2(x):
    result = x ** 2
    return result

variable = pow2(variable)
print(variable)
```

**Namespace**

| | |
|---|---|
| **variable** | **10** |
| **function**<br>**.result** | **100** |
| **variable** | **100** |

# Full Process Example 1

```python
def get_balance():
    return float(input("Enter your balance: "))

def get_withdrawal():
    return float(input("Enter amount to withdraw: "))

def process_withdrawal(balance, amount):
    if amount > balance:
        return "Insufficient funds!"
    return f"Success. Remaining: {balance - amount}"

current_balance = get_balance()
current_amount = get_withdrawal()
print(process_withdrawal(current_balance, current_amount))
```

# Full Process Example 2

```python
def get_quiz_score():
    return float(input("Quiz score: "))

def get_exam_score():
    return float(input("Exam score: "))

def compute_average(quiz, exam):
    return (quiz * 0.4) + (exam * 0.6)

def check_pass(average):
    if average >= 60:
        return "Pass!"
    return "Fail!"

quiz_score = get_quiz_score()
exam_score = get_exam_score()
average = compute_average(quiz_score, exam_score)
print("Status:", check_pass(average_score))
```

**H3**

# Sales Tracker v3

Making a more robust console tracker

# Sales Tracker v3

```python
def add(total):
    item_cost = int(input("Enter item cost: "))
    item_count = int(input("Enter item count: "))
    total_item_cost = item_cost * item_count
    return total + total_item_cost
def sub(total):
    """Remove total item cost (cost, count) from total and return"""
def show(total):
    """Print total"""

def main():
    total = 0
    running = True
    while running:
        command = input("Provide command: ")
        if command == "command 1":
            total = add(total)
        elif command == "exit":
            running = False
```

# 05

# Error Handling

Making the code secure by preparing for errors

# Possible Errors

```
1
2  divider = int(input("Number: "))
3  budget = 1_000
4  print(budget / divider)
5
6
```

# Catch Input Error

```python
1  try:
2      divider = int(input("Number: "))
3      budget = 1_000
4      print(budget / divider)
5  except ValueError:
6      print("Enter a valid number!")
```

# Catch Zero Division Error

```python
try:
    divider = int(input("Number: "))
    budget = 1_000
    print(budget / divider)
except ValueError:
    print("Enter a valid number!")
except ZeroDivisionError:
    print("Cannot pick zero")
```

# Error Raising

You can trigger errors using the `raise` keyword, followed by the error name and parentheses

```
raise Exception()
```

```
raise ValueError()
```

```
raise ValueError("Custom message here")
```

# Error Raising Example 1

```python
def withdraw(balance, amount):
    if balance < amount:
        raise ValueError("Insufficient funds")
    if amount < 0:
        raise ValueError("Withdraw amount must be positive")

    new_balance = balance - amount
    return new_balance
```

# Error Raising Example 2

```
1  try:
2      user_input = int(input("Enter Number: "))
3      if user_input < 0:
4          raise ValueError()
5
6  except ValueError:
7      print("We don't accept strings or negatives!")
```

# Final Code Execution

Given a line of code that has to run whether the code failed or not...

```
1  try:
2      print(5 / 0)
3  except:
4      print("Please don't divide by zero")
```

# Full Exception Handling

The finally keyword can be used to ensure a line of code runs no matter what happens

```python
try:
    print(5 / 0)
except:
    print("Please don't divide by zero")
finally:
    print("Code completed!")
```

**H4**

# Sales Tracker v4

Putting all of it together

# Sales Tracker v4

```python
def add(total):
    """Add item cost (cost, count) from total and return"""
def sub(total):
    """Remove item cost (cost, count) from total and return"""
def show(total):
    """Print total"""

def main():
    total = 0
    running = True
    while running:
        command = input("Provide command: ")
        if command == "command 1":
            total = add(total)
        elif command == "exit":
            running = False
```

**06**

# Lab Session

Overview of the Course and Python in General

Multiplication Table

# Multiplication Table

Ask the user for an integer input

```
1   number = int(input("Pick a number: "))
```

Print the multiplication table for that **number**

```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
…
3 x 10 = 30
```

Positive Input

# Positive Input

```
1  def positive_input():
2      number = input("Enter number: ")
3      return number
```

**Main Task:** Ask the user for a positive whole number (int) and return it as an int

**Challenges:**

- The user could provide an invalid integer input (string)
- The user could give a negative number

While the user keeps giving an invalid answer, keep asking for an input (infinite retry)

Quick Draw

# Prerequisite: Random Choice

In case we need to simulate randomness. First, put this at the top of your code.

```
1  from random import choice
```

This allows us to use the given function that returns a random item from a list

```
2  options = ["rock", "paper", "scissors"]
3  random_option = choice(options)
4  print(random_option )
```

# Recommended Project: Quick Draw

Ask the user for an input

```
1  user_choice = input("Pick a choice (rock/paper/scissors): ")
```

Make a random choice for the computer

```
2  cpu_choice = ...
```

Depending on their choices, tell if the user won, lost, or that there was a draw:
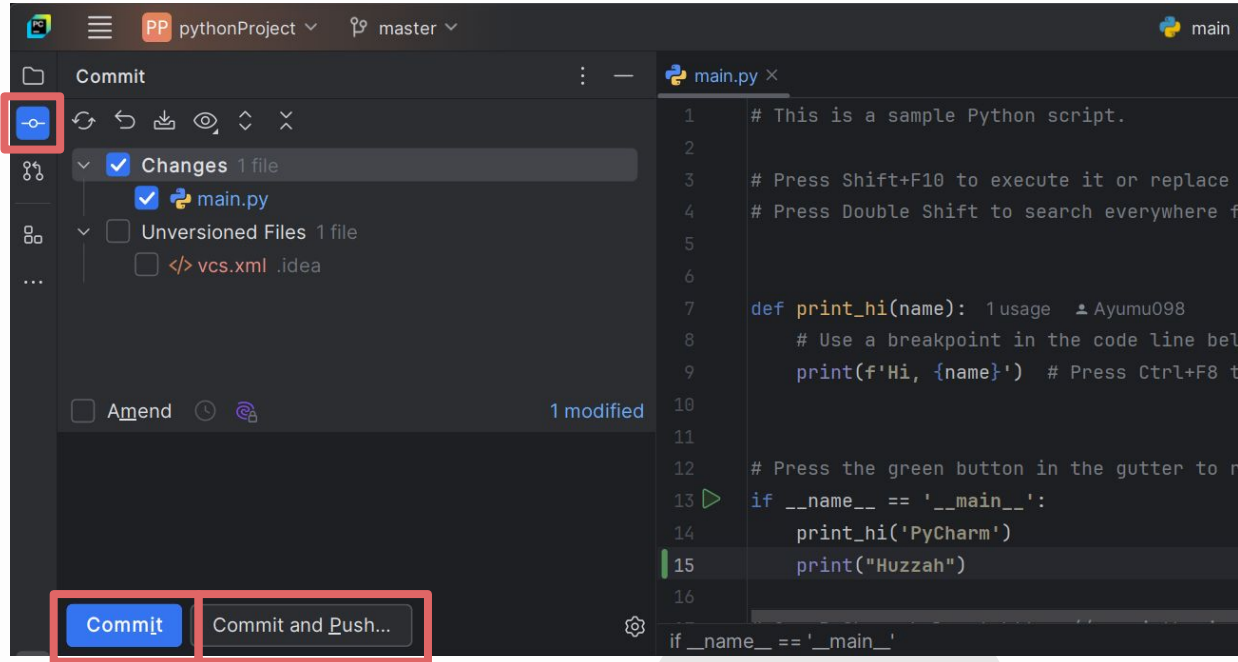
```
You win!
```

```
You Lost!
```

```
Draw!
```

Challenge: Multi-rounds

# Saving Work: Commit and Push

Select **Commit** to save locally only and **Commit and Push** to save remote as well.

# Sneak Peak

**01**

## Lists & Tuple

Ordered Group

**02**

## Dictionary & Set

Unordered Group

**03**

## String

Handling Text

**04**

## Comprehension

Iteration Shortcut

**05**

## File Handling

Data outside code

**06**

## Lab Session

Culminating Exercise

# Python: Day 01

Introduction