

Python: Day 01

Introduction to Python and Basic Syntax

Objectives



Foster a Strong Foundation

Understand the fundamental components and how to use them correctly



Develop Problem Solving Skills

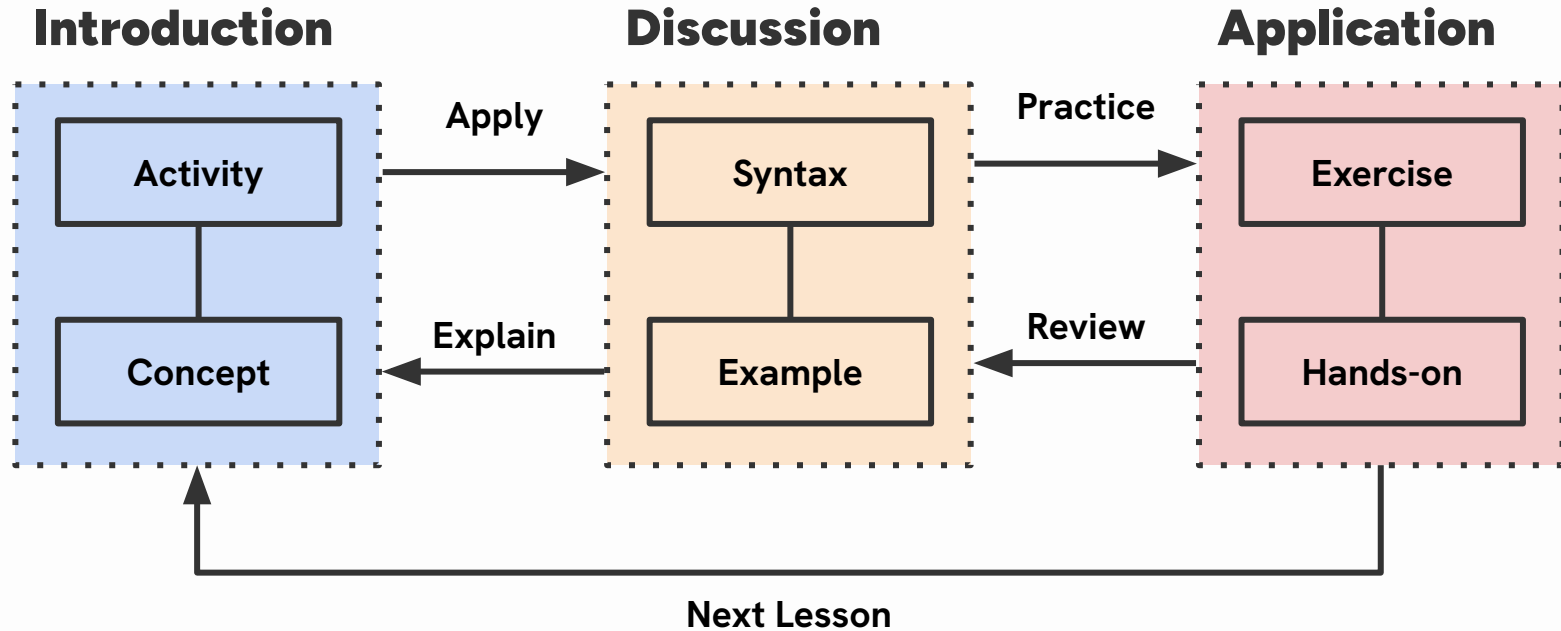
Gain practical experience through activities, exercises, and lab sessions



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's characteristics and 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

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




Python Package Index (pypi.org)

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



Or [browse projects](#)

642,013 projects

7,001,441 releases

14,411,118 files

929,410 users



Python Growth

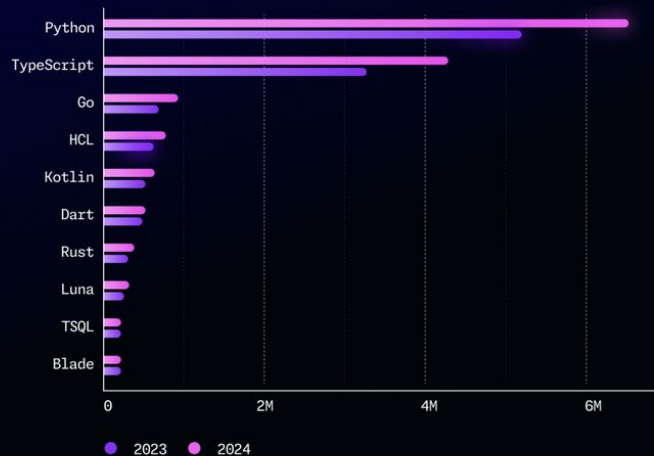
Top programming languages on GitHub

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



Top 10 fastest growing languages in 2024

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





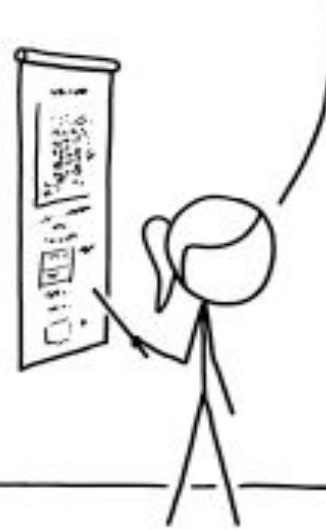
**Where can you
use Python?**

Data Science

Python is famous for data science libraries specializing in data cleaning, visualization, and modelling



OUR ANALYSIS SHOWS THAT THERE ARE THREE KINDS OF PEOPLE IN THE WORLD: THOSE WHO USE K-MEANS CLUSTERING WITH $K=3$, AND TWO OTHER TYPES WHOSE QUALITATIVE INTERPRETATION IS UNCLEAR.



Machine Learning

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

 PyTorch

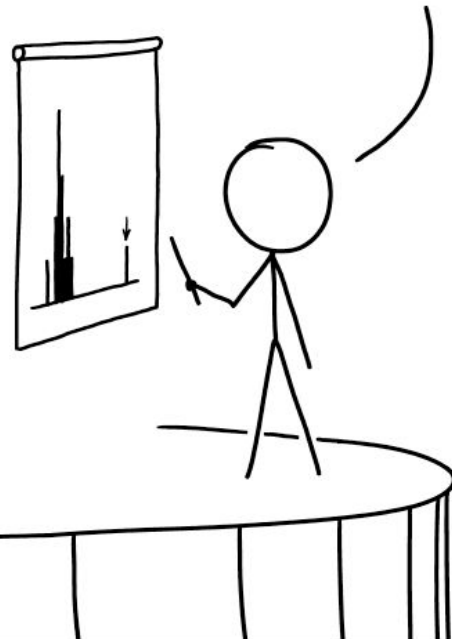
 TensorFlow

 spaCy

 OpenCV

DESPITE OUR GREAT RESEARCH RESULTS, SOME HAVE QUESTIONED OUR AI-BASED METHODOLOGY.

BUT WE TRAINED A CLASSIFIER ON A COLLECTION OF GOOD AND BAD METHODOLOGY SECTIONS, AND IT SAYS OURS IS FINE.



xkcd

Web Development

Alternatives to the traditional web tech stack include libraries and frameworks that Python can provide.



THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:

RUNNING NPM INSTALL



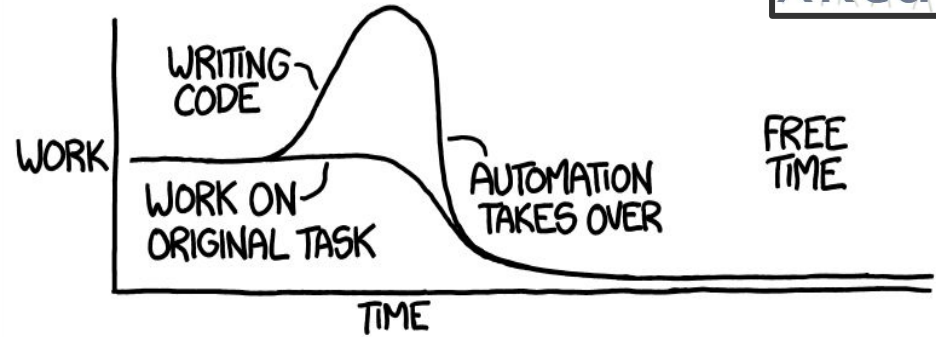
Automation

The key use of programming is to automate the boring tasks to make it faster and easier.

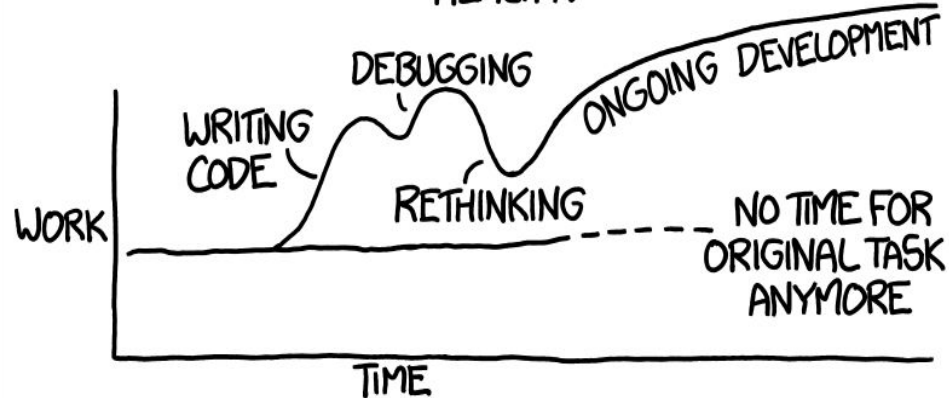


"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"

THEORY:

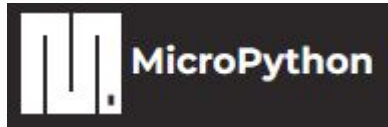


REALITY:



Specialist Fields

Python is a common entry-point for specialists to build and process their knowledge base.



WHEN A USER TAKES A PHOTO,
THE APP SHOULD CHECK WHETHER
THEY'RE IN A NATIONAL PARK...

SURE, EASY GIS LOOKUP.
GIMME A FEW HOURS.

... AND CHECK WHETHER
THE PHOTO IS OF A BIRD.

I'LL NEED A RESEARCH
TEAM AND FIVE YEARS.



IN CS, IT CAN BE HARD TO EXPLAIN
THE DIFFERENCE BETWEEN THE EASY
AND THE VIRTUALLY IMPOSSIBLE.

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

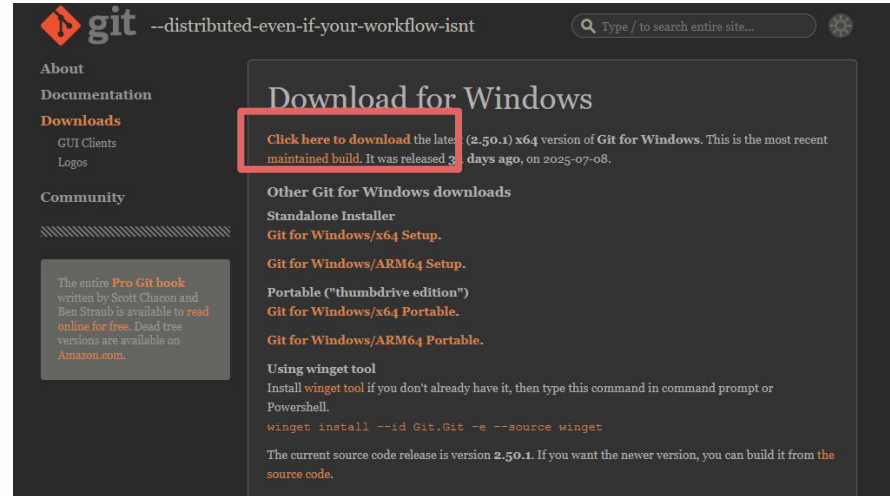
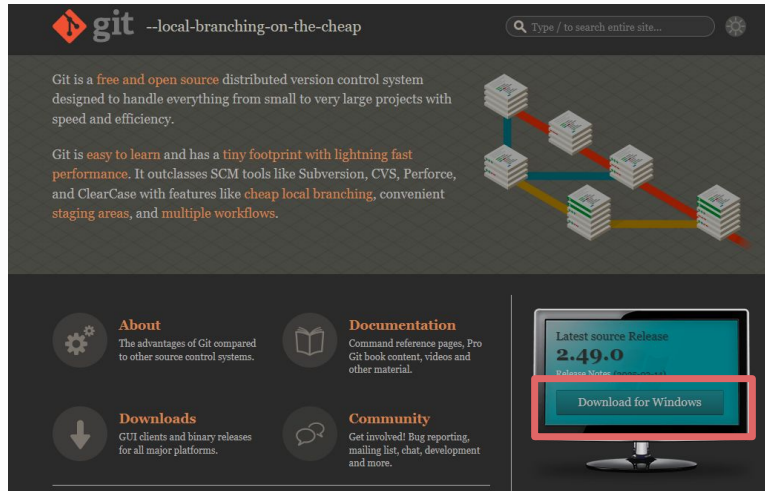
First version released in December 2008, made to **upgrade performance, add extra features, and improve clarity** without being backwards compatible

Setup Tools

Full instructions for your initial development setup

Step 1: Download Git

Go to <https://git-scm.com/> and select the download option



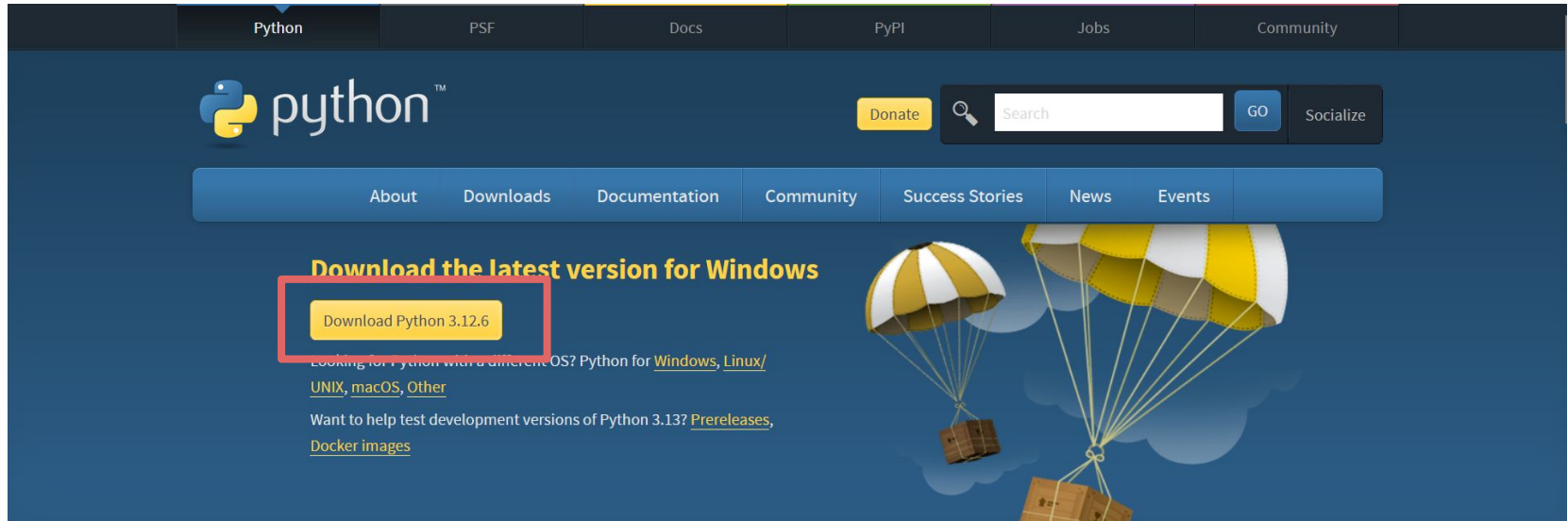
Step 2: Git Installation Setup

Run the git installer and use the default options



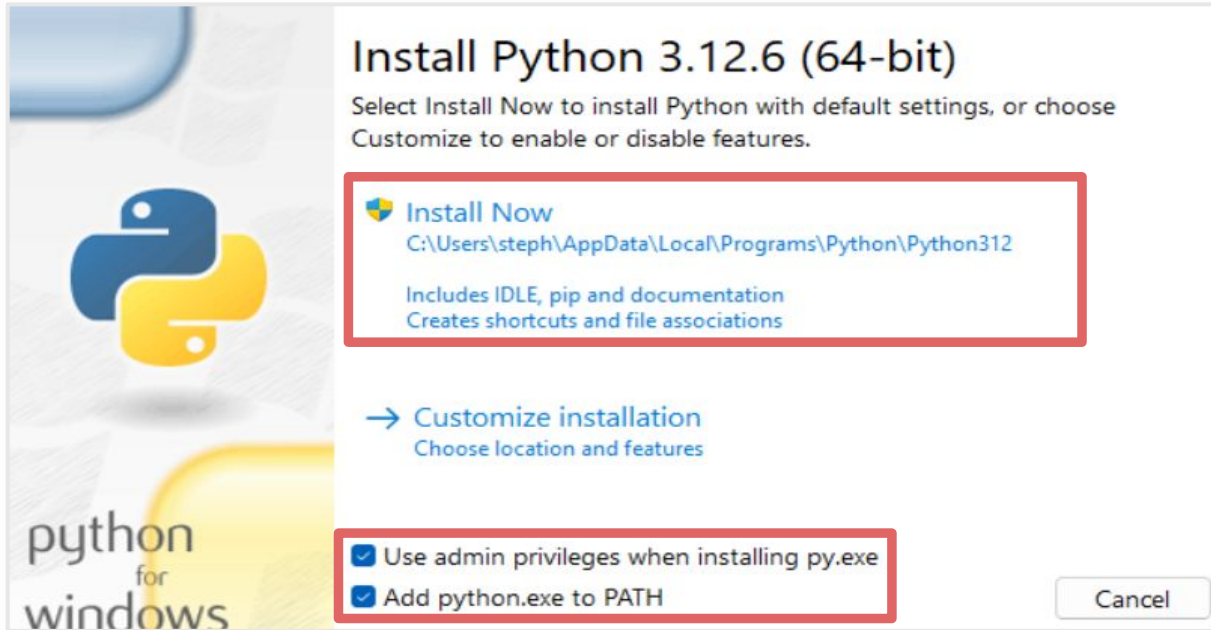
Step 3: Download Python Downloader

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.

PyCharm JetBrains IDEs

Use Cases ▼ EAP What's New Features ▼ Learn ▼ Pricing Download

Windows macOS Linux

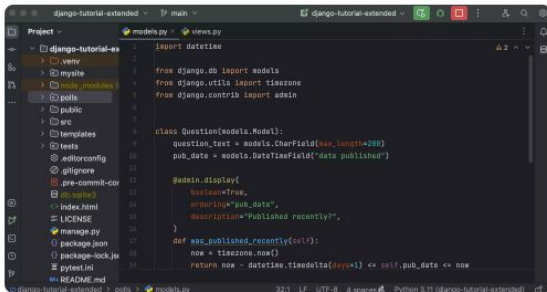
 **PyCharm** Unified Product

The only Python IDE you need

Download

 .exe (Windows) ▼

Free forever, plus one month of Pro included



```
Project -> django-tutorial-extended
models.py
views.py

from django.db import models
from django.utils import timezone
from django.contrib import admin

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    @admin.display(
        description='pub_date',
        description='Published recently?',
    )
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

Version: 2025.1.1.1
Build: 251.25410.159
14 May 2025

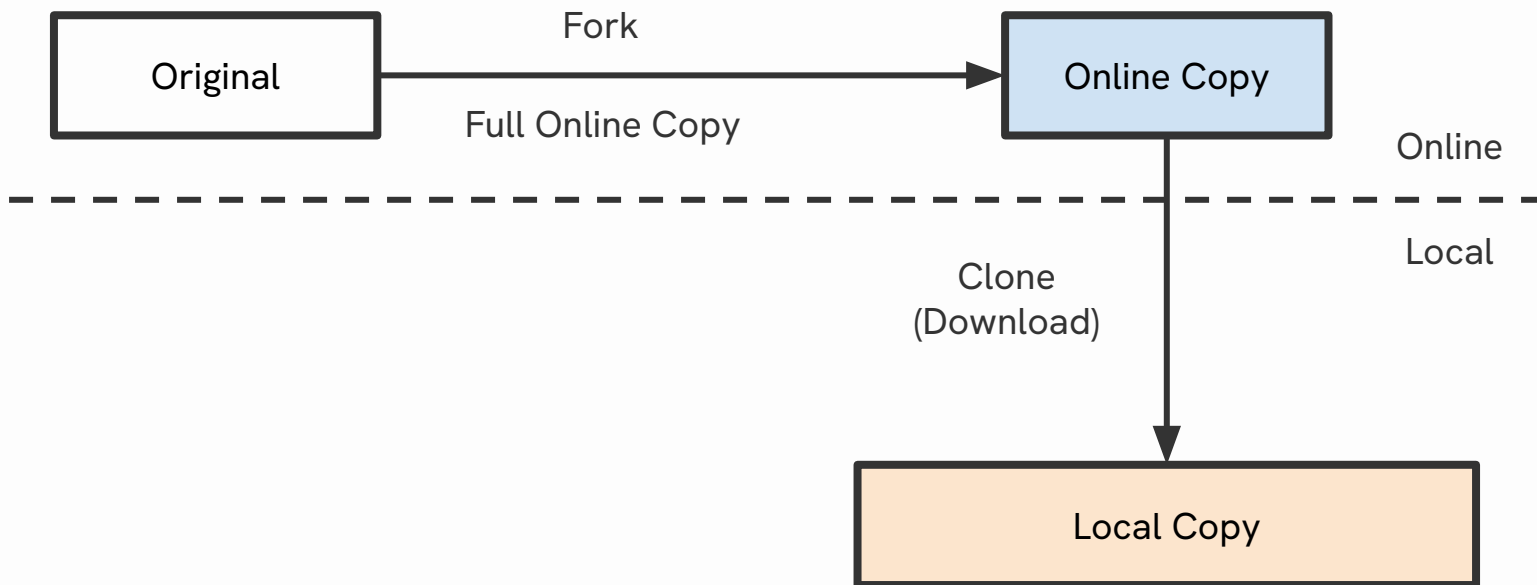
[System requirements](#)
[Installation instructions](#)

[Other versions](#)
[Third-party software](#)

Repository

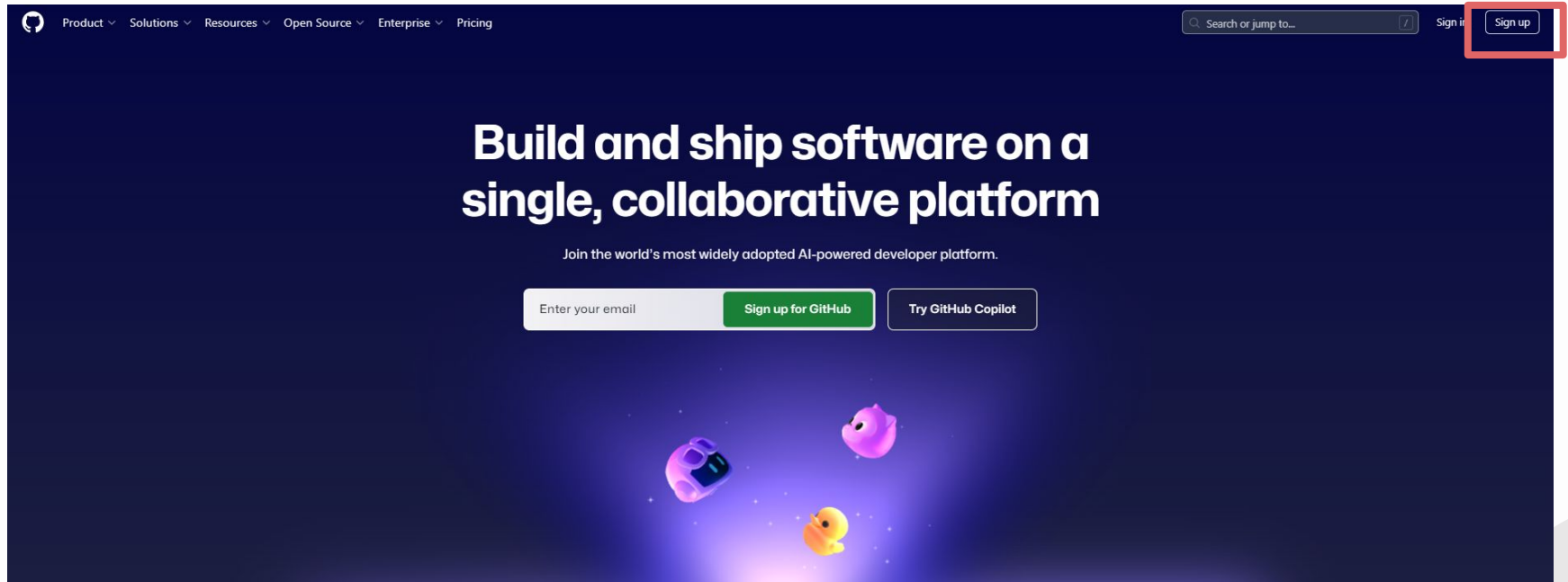
Securing access to the course materials

Repository System



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 github.com/Ayumu098/python-bootcamp and select the **Fork** button on the upper right

The screenshot shows the GitHub interface for the repository 'python-bootcamp' by user 'Ayumu098'. The repository is public and has 0 stars and 0 forks. The 'Fork' button is highlighted with a red box. The repository contains a single commit 'Initial commit' by 'Ayumu098' 4 months ago. The commit details show a list of files: .idea, samples, slides, and zip, all added in the initial commit. The right sidebar shows the repository description: 'This repository contains original learning materials and code snippets for a four-day Python Programming Fundamentals course.' and links to the README, MIT license, Activity, and 0 stars.

python-bootcamp Public

Unpin Unwatch 1 Fork 0 Star 0

master 1 Branch 0 Tags

Go to file Add file <> Code

About

This repository contains original learning materials and code snippets for a four-day Python Programming Fundamentals course.

Readme MIT license Activity 0 stars 1 watching

Step 3: Setup Fork Settings

Select the **Create Fork** button

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Required fields are marked with an asterisk ().*

Owner *

Repository name *

Choose an owner ▾

/ python-bootcamp

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

This repository contains original learning materials and code snippets for a four-day Python Programming Fun

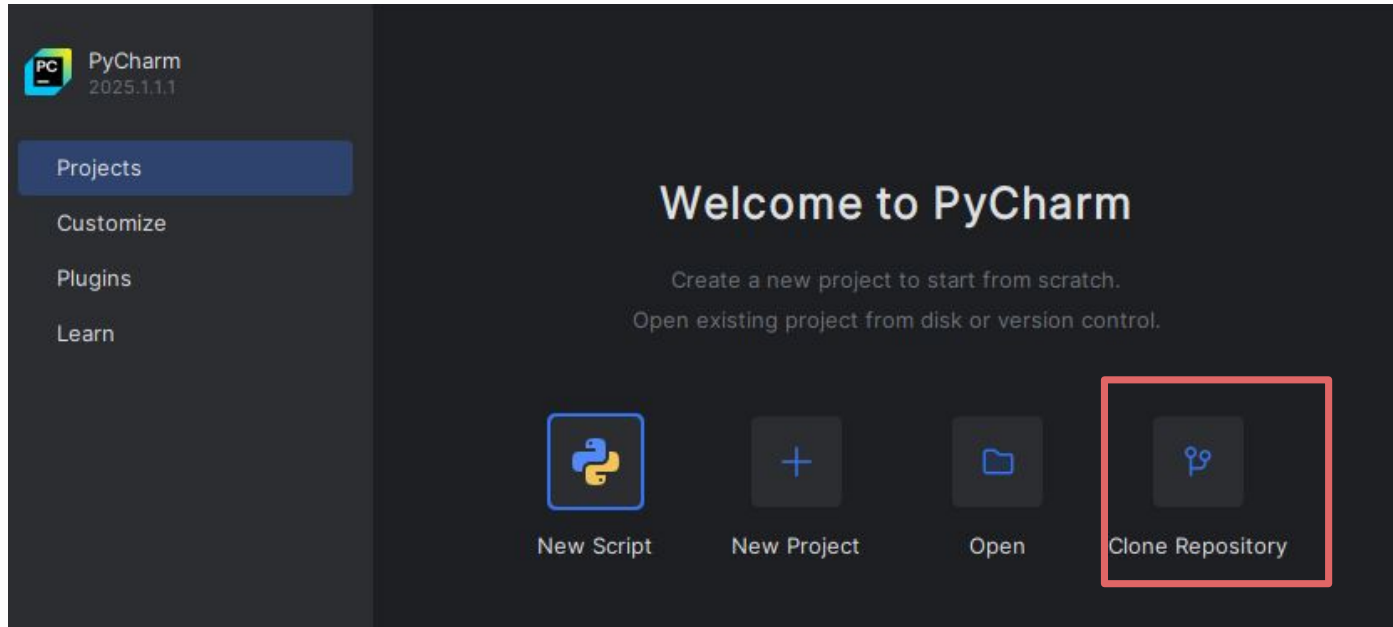
☒ Copy the `master` branch only

Contribute back to Ayumu096/python-bootcamp by adding your own branch. [Learn more.](#)

Create fork

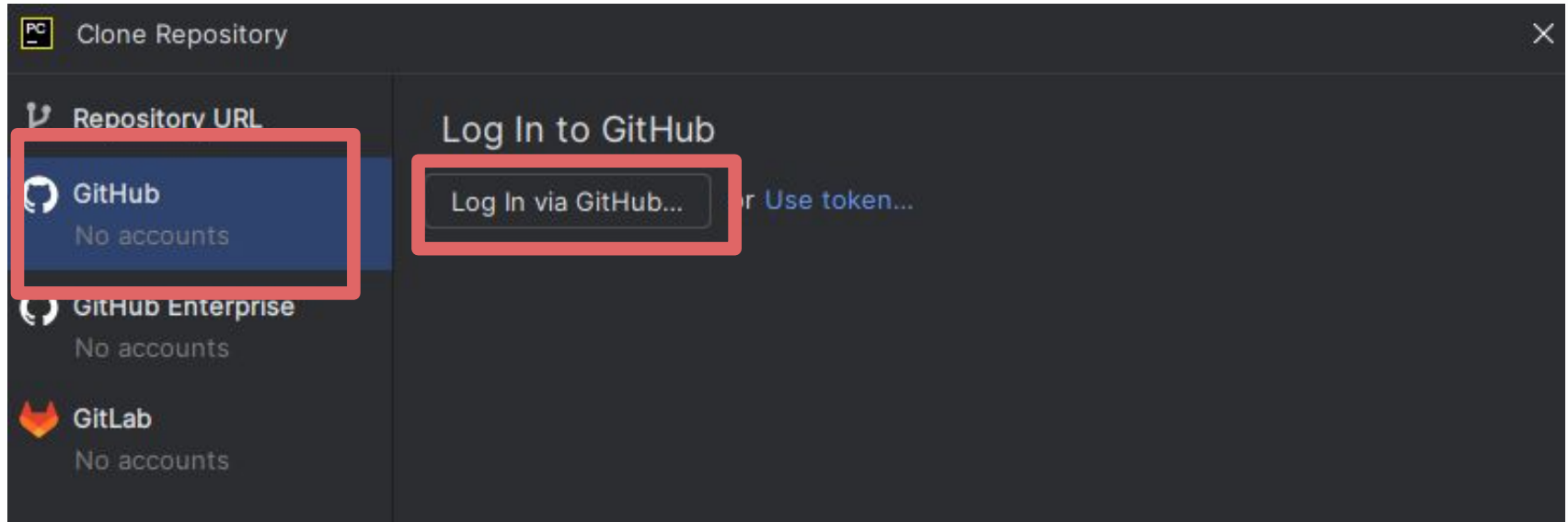
Step 4: Clone Repository

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*

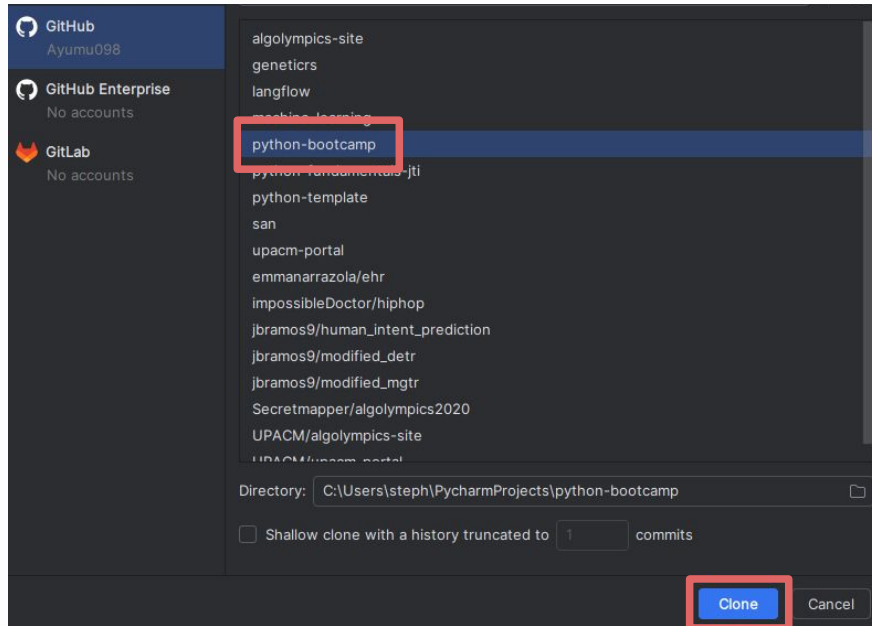


Please continue only if this page is opened from a [JetBrains IDE](#).

Authorize in GitHub

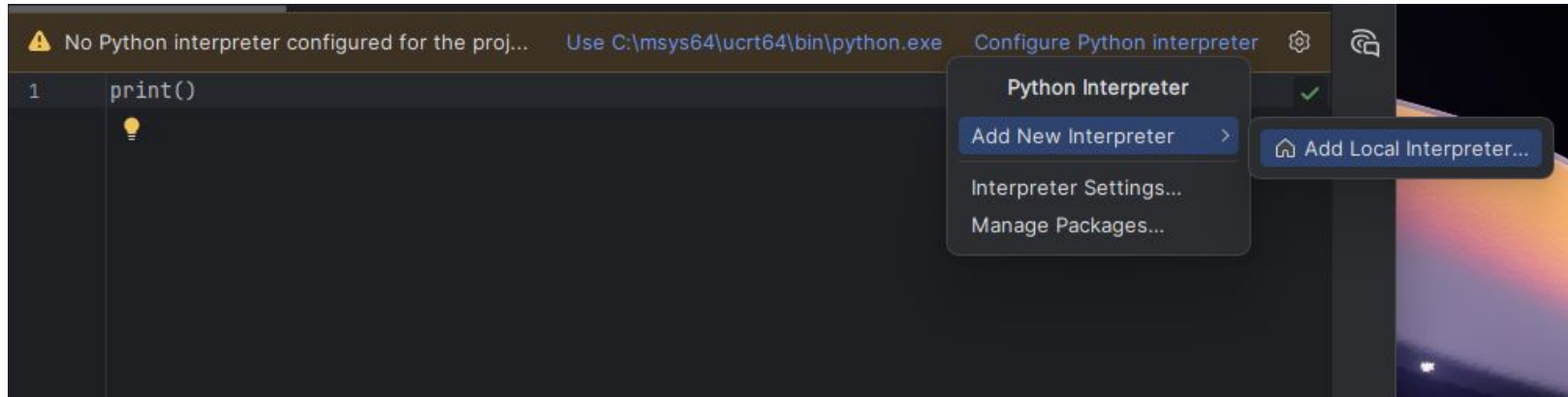
Step 7: Select Repository

The window shows all your repositories. Select *python-bootcamp* and **Clone**.



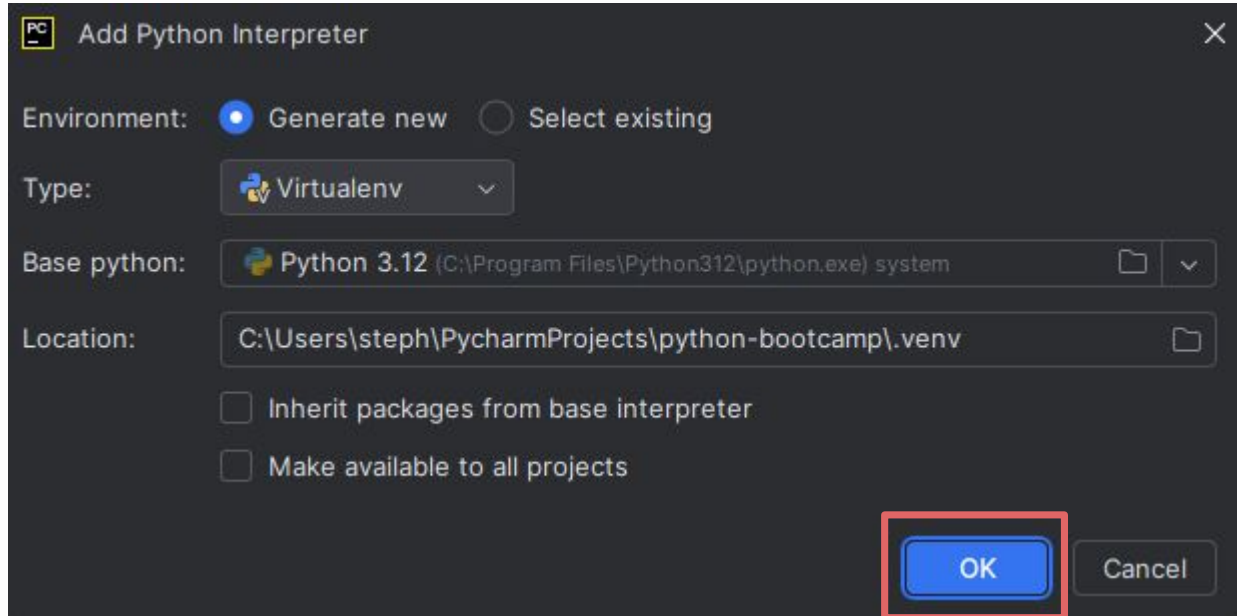
Step 8: Add Python Interpreter

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**.

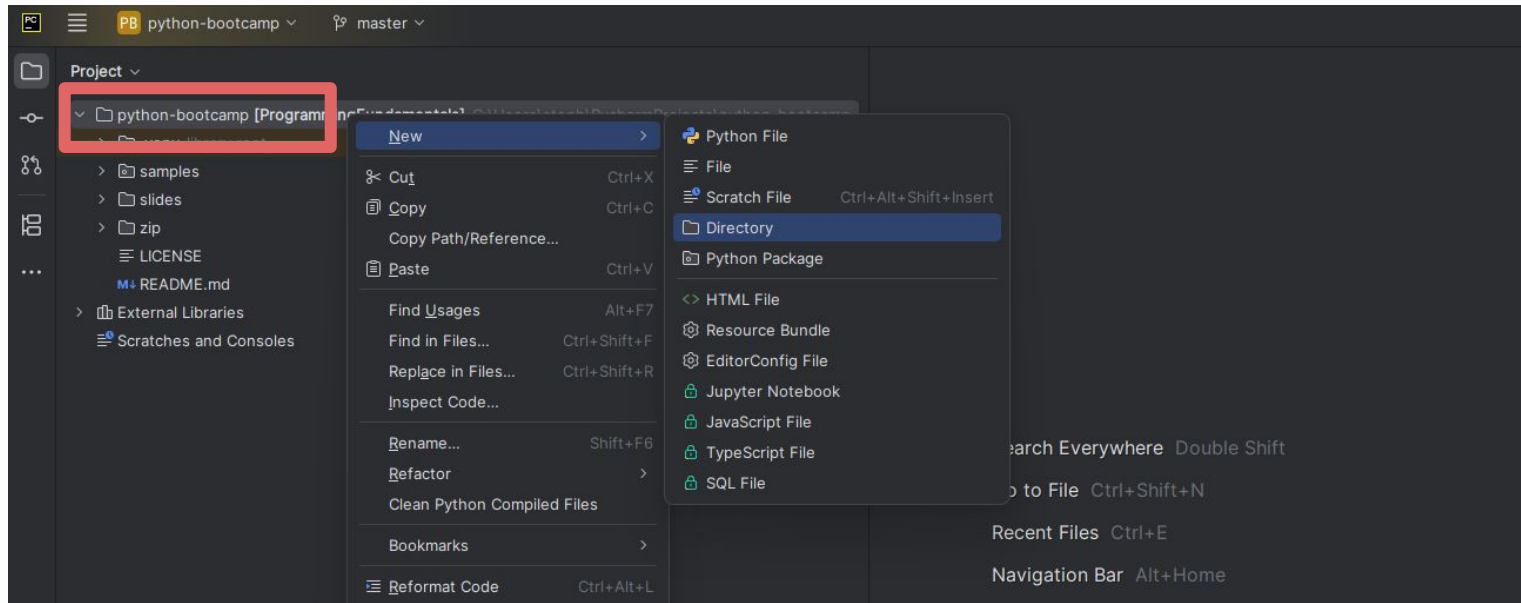


Hello World

A journey of a thousand miles begins with a single step

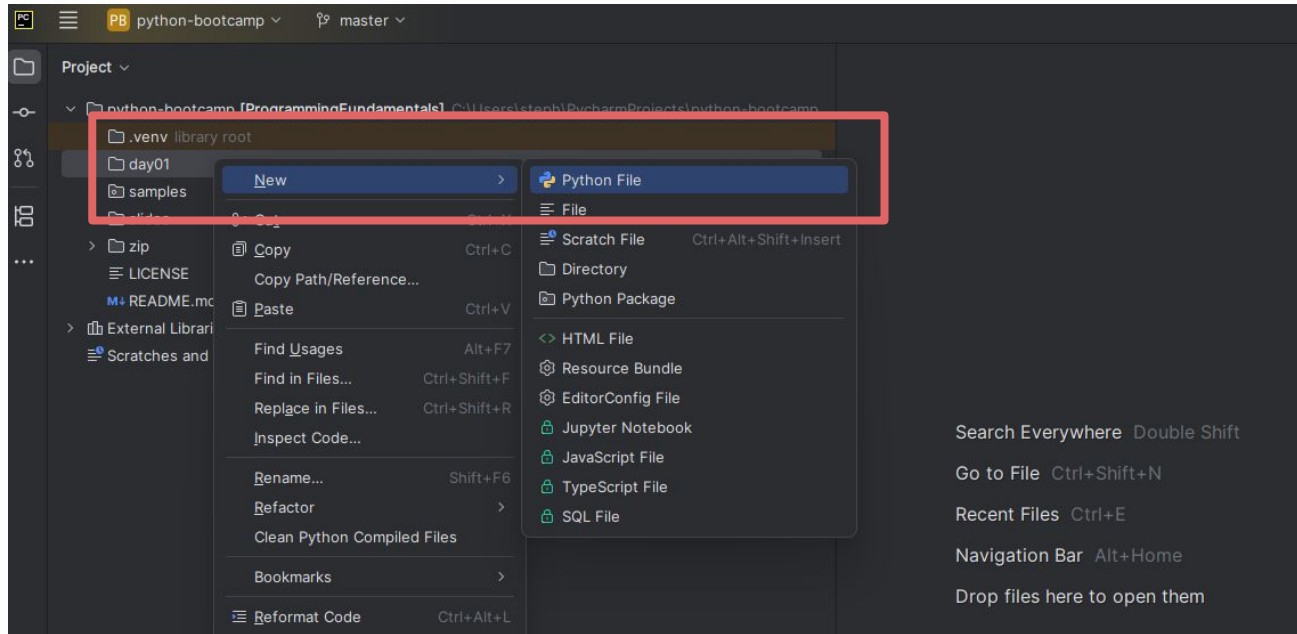
Create a New Folder

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

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

Quick Exercise: Hello World

Print the following in the console:

01_hello.py

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

Comments for Documentation

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

```
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!")
```

02

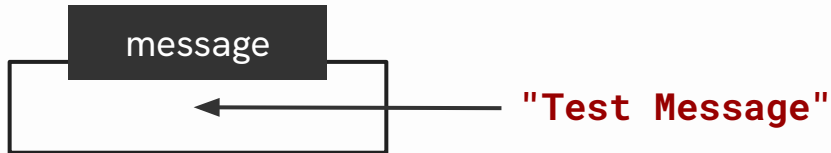
Basics

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"
```



Variable Printing

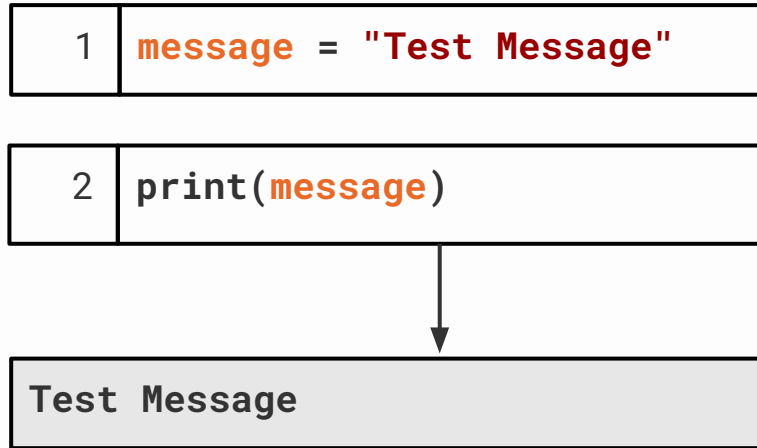
Variables can also be displayed on the console with the **print** function

1	<code>message = "Test Message"</code>
---	---------------------------------------

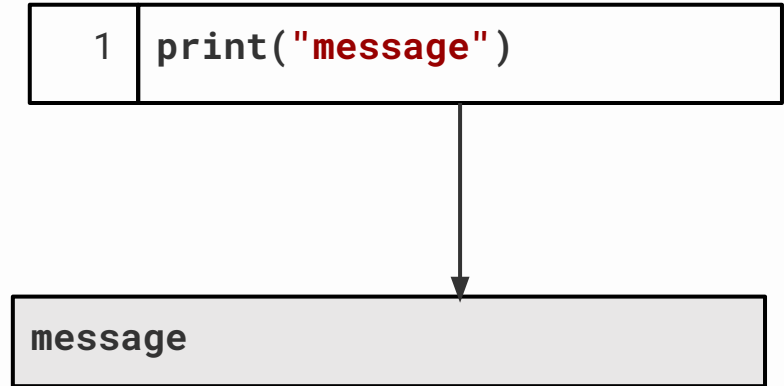
2	<code>print(message)</code>
---	-----------------------------

Variables and Text

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



≠



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:

01_hello.py

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

New Code:

01_hello.py

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

Variable Change

Updating existing variables

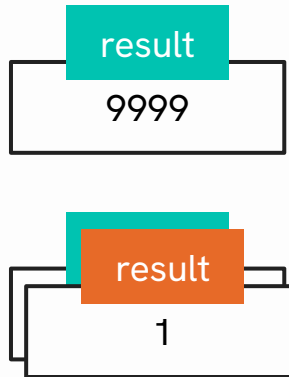
Variable Reassignment

Variables can be changed by using the same variable name

```
result = 9999  
print(result)
```

```
result = 1  
print(result)
```

9999
1



Variable Evaluation

Python evaluates code **top to bottom, right to left**



age = 30

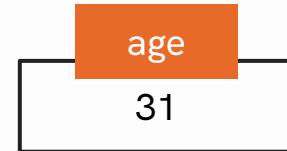
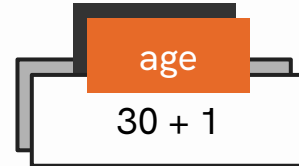
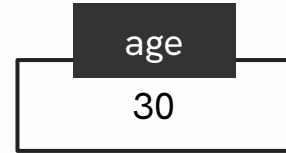


age = **age** + 1



age = 31

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



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)

```
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

02_counter.py

```
1 counter = 0
2 print("Counter:", counter)
3
4 # Point up: Add one to the counter
5 # Code here
6 print("Counter:", counter)
7
8 # Bonus: Multiply the score by 10
9 # Code here
10 print("Counter:", counter)
11
12 # Penalty: Decrease the score by 4
13 # Code here
14 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

03_wishlist.py

```
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)
```

Quick Exercise: Expense Tracker

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)
```

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

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

Strings and Numbers

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

1	<code>number = "123"</code>
2	<code>number = number + 1</code>

≠

1	<code>number = 123</code>
2	<code>number = number + 1</code>

Integer Type Conversion

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

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



```
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 → 1, False → 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

```
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 = Calculate the sum of the numbers
12 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**

```
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

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

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:

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

```
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

```
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
<code>result = result + 5</code>	<code>result += 5</code>
<code>result = result * 10</code>	<code>result *= 10</code>
<code>message = message + "World"</code>	<code>message += "World"</code>

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)

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

Quick Exercise: Score System (v2)

02_counter.py

```
1 counter = 0
2 print("Counter:", counter)
3
4 # Point up: Add one to the counter
5 # Change your code in this line
6 print("Counter:", counter)
7
8 # Bonus: Multiply the score by 10
9 # Change your code in this line
10 print("Counter:", counter)
11
12 # Penalty: Decrease the score 4
13 # Change your code in this line
14 print("Counter:", counter)
```

String Formats

Combine strings and variables conveniently

String Placeholder

```
1 # Message Template
2 message = "Hello {}! Nice to meet you!"
3 print(message)
4
5 # Use Template
6 formatted_message = message.format("Juan")
7 print(formatted_message)
```

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

String Placeholder (Repeated Use)

```
1  # Message Template
2  message = "Hello {}! Nice to meet you!"
3  print(message)
4
5  # Use Template
6  formatted_message = message.format("Juan")
7  print(formatted_message)
8
9  # Use Template (again)
10 new_message = message.format("Jesse")
11 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.

```
1 message = "Hello {}. Your nickname is {}"  
2  
3 name = input("Enter name: ")  
4 nickname = input("Enter nickname: ")  
5  
6 formatted_message = message.format(name, nickname)  
7 print(formatted_message)
```


Multiple String Formatting (Named)

Placeholders can be given a variable name to make assignment easier

```
1 message = "Hello {first}. Your nickname is {second}"
2
3 name = input("Enter name: ")
4 nickname = input("Enter nickname: ")
5
6 formatted_message = message.format(first=name, second=nickname)
7 print(formatted_message)
```

Quick Exercise: Price Post

06_price_post.py

```
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)
```

H1

Sales Tracker

Quick practice of all the concepts discussed so far

Quick Exercise: Sales Tracker

07_sales_tracker.py

```
1 # Ask the cost and pax or count for three separate items
2 item_cost_1 = Let the user enter a number
3 item_count_1 = Let the user enter a number
4
5 item_cost_2 = Let the user enter a number
6 item_count_2 = Let the user enter a number
7
8 item_cost_3 = Let the user enter a number
9 item_count_3 = Let the user enter a number
10
11 # Calculate the total
12 total = 0
13 print(total)
```

03

Control Flow

Providing logic to data processing

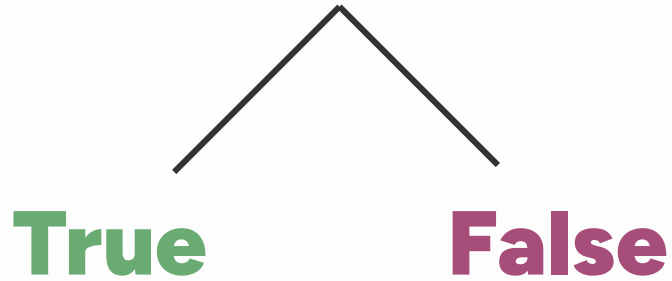
Relations

Checking if two values are related to each other



The result can be stored

result = number_1 > number_2







What are the possible results?

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

```
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

```
1 x = int(input("Enter number: "))
2
3 print("Exclusive Range")
4 print(3 < x < 20)
5
6 print("Equal or Greater than 3 and Less than 20")
7 print(3 <= x < 20)
8
9 print("Greater than 3 and Less than or Equal to 20")
10 print(3 < x <= 20)
11
12 print("Inclusive Range")
13 print(3 <= x <= 20)
```


Quick Exercise: Valid Score

09_valid_score.py

```
1  # Range minimum and maximum bounds
2  min_number = 0
3  max_number = 100
4
5  # Enter user input
6  number = Enter number
7
8  # Notify user if the number is a valid score
9  valid_score = None
10 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

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:", correct_password_given )
```

Conditionals

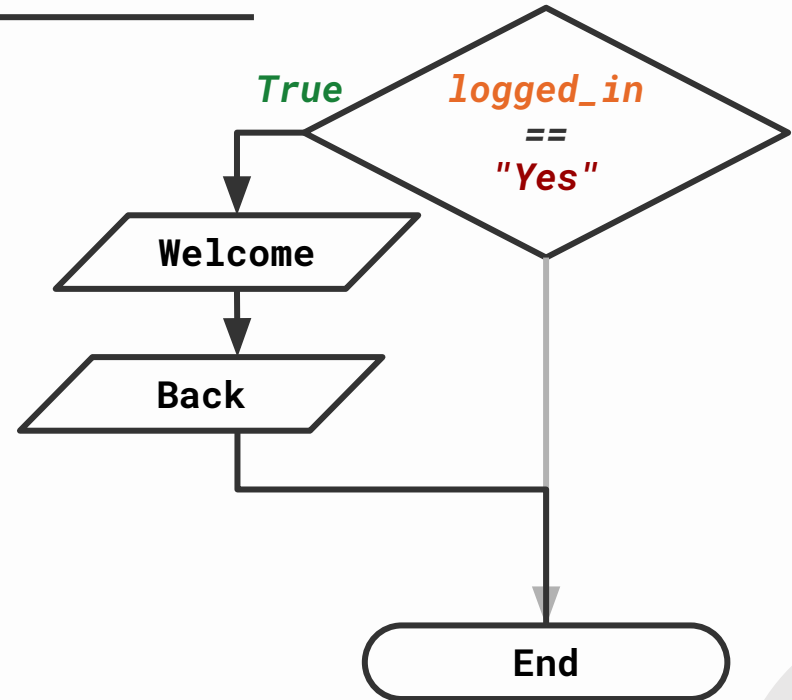
Control when code executes

If Statement - True

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

Welcome
Back
End

Input: "Yes"

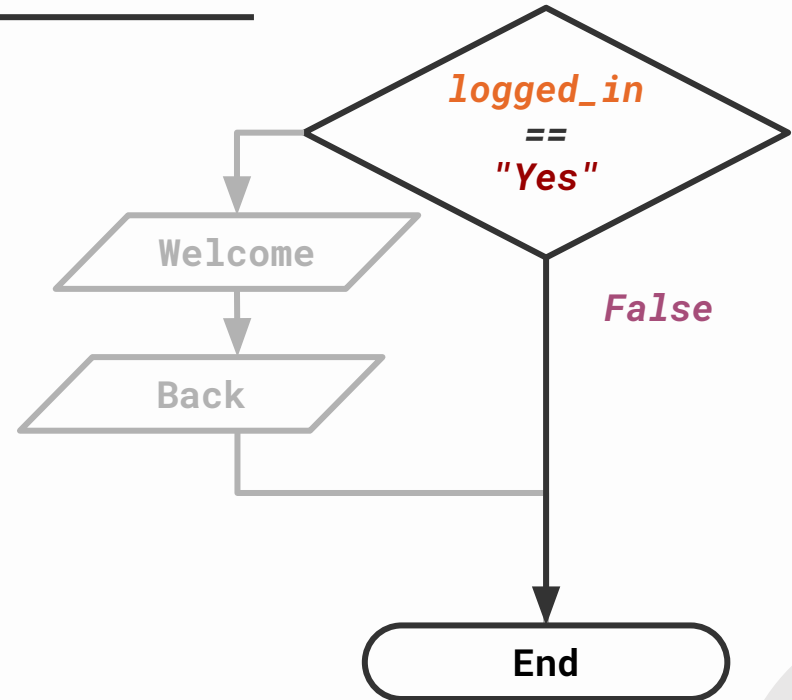


If Statement - False

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

```
Welcome
Back
End
```

Input: "No"



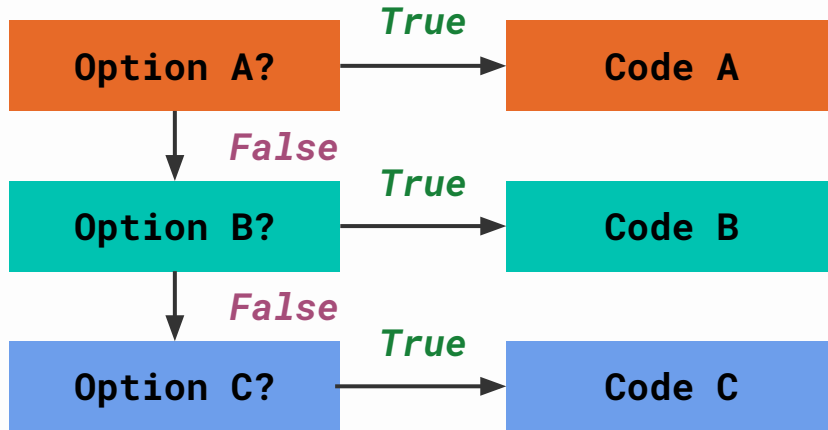
Quick Exercise: Login System (v2)

10_login_system.py

```
1  # Expected password (you can change the value)
2  correct_password = "pass"
3
4  # Enter 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")
```


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

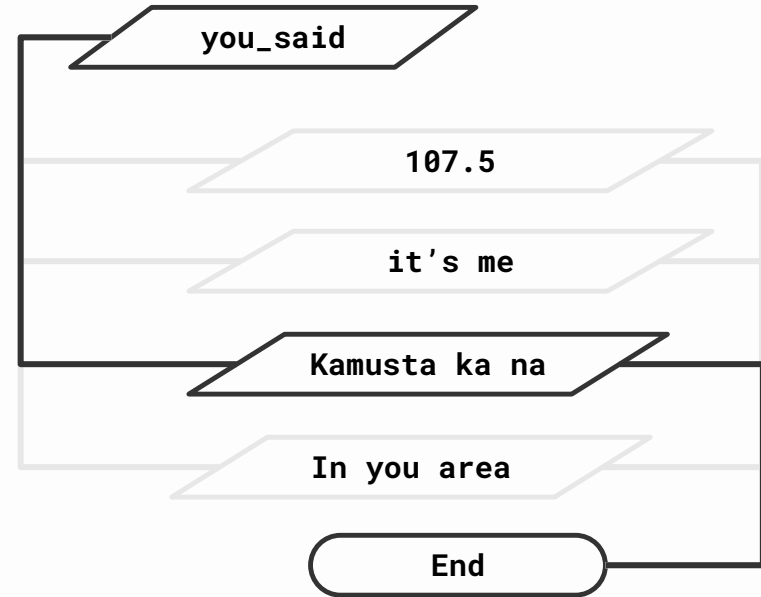


```
if condition_1:  
    """Code A"""  
  
elif condition_2:  
    """Code B"""  
  
elif condition_3:  
    """Code C"""
```

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



Elif Statement Example 02

```
1 battery = int(input("Battery percentage: "))
2
3 if battery >= 80:
4     print("Full Battery")
5 elif battery >= 40:
6     print("Good Battery")
7 elif battery >= 15:
8     print("Low Battery")
9 elif battery > 0:
10    print("Critically Low Battery")
```

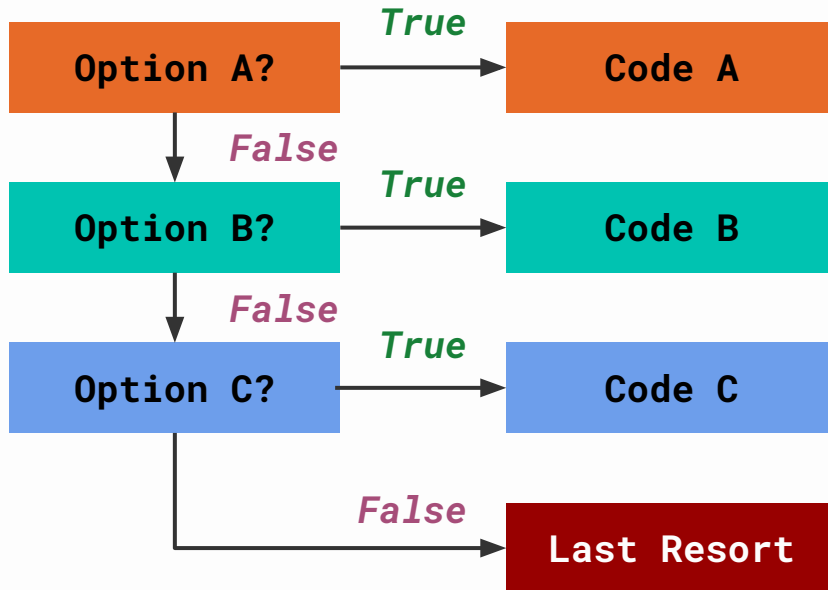
Quick Exercise: Greetings

11_traffic_lights.py

```
1  # Ask the user input for a color
2  color_input = input("Please enter a color: ")
3
4  # Print the following depending on the color input
5  # "green"    -> print "Go"
6  # "yellow"   -> print "Wait..."
7  # "red"      -> print "Stop"
```

Else Statement (Last Resort)

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



```
if condition_1:
    """Code A"""
elif condition_2:
    """Code B"""
elif condition_3:
    """Code C"""
else:
    """Last Resort"""
```

Else Statement

The else statement is often used to notify on unexpected issues in the input

```
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 that song!")
```

Quick Exercise: Traffic Lights (version 2)

11_traffic_lights.py

```
1  # Ask the user input for a color
2  color_input = input("Please enter a color: ")
3
4  # Print the following depending on the color input
5  # if green
6  #     -> "Go"
7  # elif yellow
8  #     -> "Wait..."
9  # elif red
10 #     -> "Stop"
11 # else
12 #     -> "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

```
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

```
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

```
1 age = int(input("Enter age: "))
2 if age >= 18:
3     print("Old enough to watch movie")
4 else:
5     print("Too young to watch movie")
```

```
1 balance = 150
2 price = 200
3
4 if balance >= price:
5     print("Payment successful")
6 else:
7     print("Insufficient funds")
```

Quick Exercise: Login System (v3)

10_login_system.py

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

Logical Operators

Simplifying conditionals

And Operator

Use the **and** operator to restrict conditions

```
1 applied = True
2 has_skill = True
3 has_experience = True
4
5 if applied and has_skill and has_experience:
6     print("You're hired")
```

And Operator Example

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

```
1 money = float(input("Enter money: "))
2 stock = int(input("Enter stock: "))
3
4 if money >= 100 and stock > 0:
5     print("You can buy the item!")
6 else:
    print("You can't buy the item")
```

Quick Exercise: Login System (v4)

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

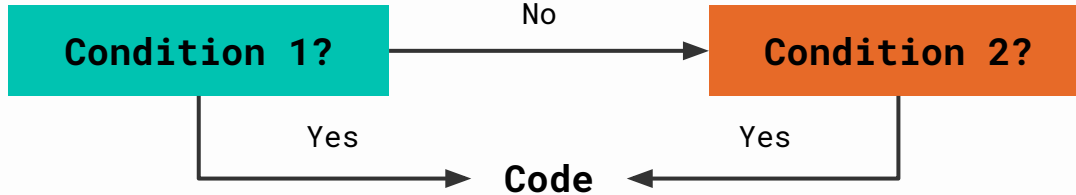
Use the **or** operator to add alternative conditions

```
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

```
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

```
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

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

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

```
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

12_bookmarks.py

```
1 # Define a list of your favorite websites  
2 websites = ["facebook.com", "youtube.com"]  
3  
4 # Print the entire list of websites  
5 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)
```

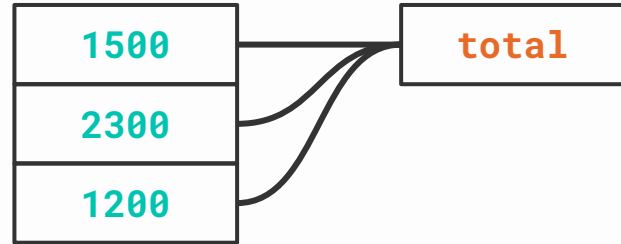
```
Alert: Battery low  
Alert: New message  
Alert: New Update
```

For Loop Example 02: Aggregation

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

```
1 expenses = [1500, 2300, 1200]
2 total = 0
3
4 for amount in expenses:
5     total += amount
6
7 print("Total expenses:", total)
```

Total expenses: 5000

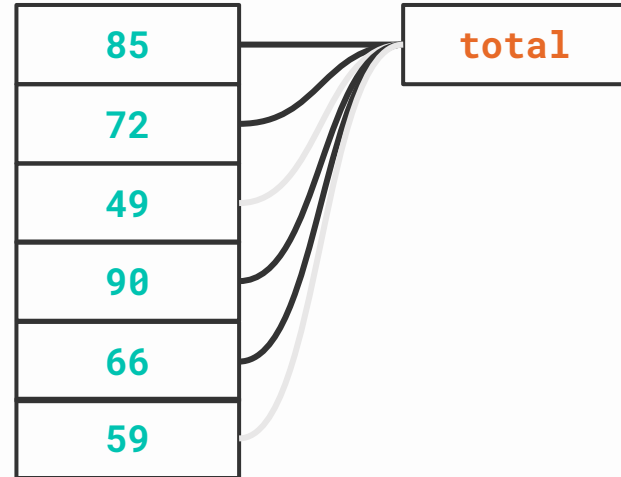


For Loop Example 03: Counting

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



Quick Exercise: Bookmarks (v2)

12_bookmarks.py

```
1 # Define a list of your favorite websites (add or change below)
2 websites = ["facebook.com", "youtube.com"]
3
4 # Print the entire list of websites (one at a time)
5 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

```
1 # Long Message
2 message = "This is a very long message that's hard to type"
3
4 # Print the message eleven times
5 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

```
1 # Ask the user for a number  
2 end = int(input("Enter number: "))  
3  
4 # Print the numbers 0 to end  
5 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

```
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

```
1  running = True
2  while running:
3      command = input("Provide command: ")
4      if command == "command 1":
5          print("command 1 done")
6      elif command == "command 2":
7          print("command 2 done")
8      elif command == "command 3":
9          print("command 3 done")
10     elif command == "exit":
11         running = False
```


Quick Exercise: Running Balance

16_running_balance.py

```
1 total = 0
2 running = True
3 while running:
4     command = input("Provide command: ")
5
6     if command == "add":
7         # Ask for number, add to total, and print
8     if command == "sub":
9         # Ask for number, subtract to total, and print
10    elif command == "exit":
11        running = False
```

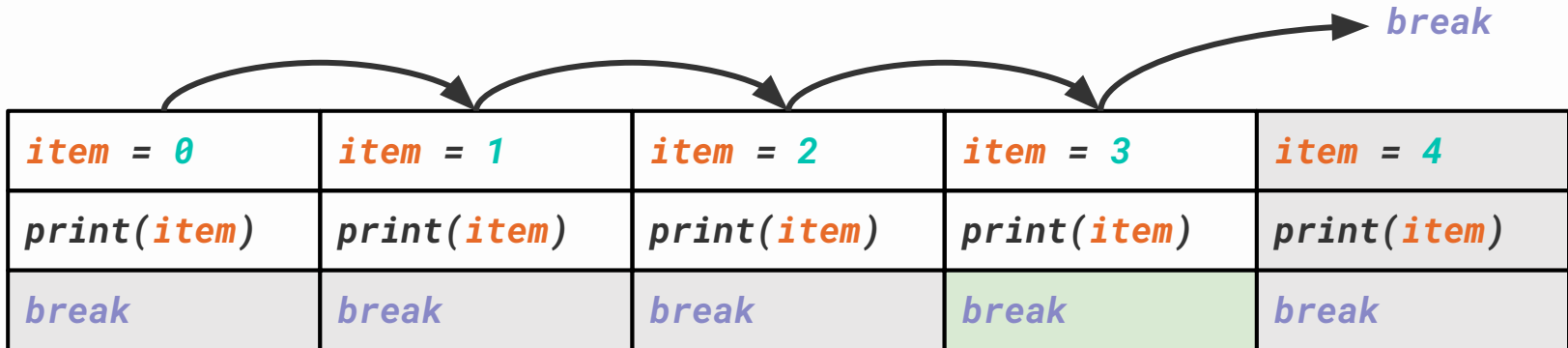
Loop Control

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
```



While Loop: Password Attempt

```
1 max_attempt = 3
2 correct_password = "pass"
3
4 for attempt in range(max_attempt):
5     password = input("Enter password: ")
6     if password == correct_password:
7         print("Access granted")
8         break
```

Quick Exercise: Search

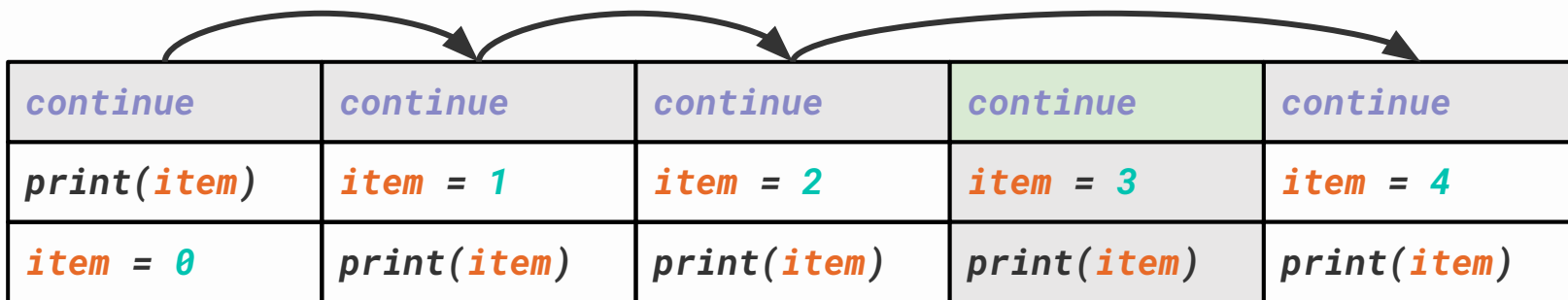
17_search.py

```
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

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



Quick Exercise: Skip Range

18_skip_range.py

```
1 for item in range(100):  
2     # Change code to skip printing numbers 20 to 80.  
3     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	<code>def extra():</code>
2	<code> print("Extra Line 1")</code>
3	<code> print("Extra Line 2")</code>
4	
5	<code>print("First Line")</code>
6	<code>extra()</code>
7	<code>print("Second Line")</code>

1	<code>print("First Line")</code>
2	
3	<code>print("Extra Line 1")</code>
4	<code>print("Extra Line 2")</code>
5	
6	<code>print("Second Line")</code>

First Line
Extra Line 1
Extra Line 2
Second Line

Simple Function Declaration

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

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

```
3 greet()
```

Quick Exercise: Line Generator

19_line_generator.py

```
1  """
2  Create a function line_generator that prints the following:
3      Line 1
4      Line 2
5      Line 3
6  """
7
8  # Use the function once
9  line_generator()
```

```
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)

19_line_generator.py

```
1  """
2  Create a function line_generator that has a parameter number
3  and prints the following:
4      Line 1
5      Line 2
6      ...
7      Line number
8  """
9
10 # Use the function once
11 line_generator(4)
```

Multiple Input Declaration

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

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

```
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

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

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

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

Optional Parameter (Overriding)

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

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

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

Quick Exercise: Product (v2)

20_product.py

```
1 def product():
2     """Takes three inputs (or two inputs) and print the product"""
3
4     product(1, 1, 1) # 1
5     product(1, 2, 3) # 6
6     product(2, 5, 10) # 100
7     product(3, 3) # 9
8     product(2, 5) # 10
```

Function Returns

Simplifying calculations and data handling

Return Value

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

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

```
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

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

3

```
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

```
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

```
1 def product():
2     """Takes three inputs (or two inputs) and return"""
3
4     product(1, 1, 1) # 1
5     product(1, 2, 3) # 6
6     product(2, 5, 10) # 100
7     product(3, 3) # 9
8     product(2, 5) # 10
```


Function Scope

Determining variable lifetime

Functions can read outside

Function can detect and print variables outside of it

```
outer_variable = 10

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

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

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

Namespace

outer_variable	10
----------------	----

Functions can't write outside

Function can't edit variables outside normally

```
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 .outer_variable	999

Functions update using return

Function can interact outside using a return statement

```
variable = 10

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

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

Namespace



variable	10
function .result	100
variable	100

Full Process Example 1

```
1 def get_balance():
2     return float(input("Enter your balance: "))
3
4 def get_withdrawal():
5     return float(input("Enter amount to withdraw: "))
6
7 def process_withdrawal(balance, amount):
8     if amount > balance:
9         return "Insufficient funds!"
10    return f"Success. Remaining: {balance - amount}"
11
12 current_balance = get_balance()
13 current_amount = get_withdrawal()
14 print(process_withdrawal(current_balance, current_amount))
```

Full Process Example 2

```
1 def get_quiz_score():
2     return float(input("Quiz score: "))
3
4 def get_exam_score():
5     return float(input("Exam score: "))
6
7 def compute_average(quiz, exam):
8     return (quiz * 0.4) + (exam * 0.6)
9
10 def check_pass(average):
11     if average >= 60:
12         return "Pass!"
13     return "Fail!"
14
15 quiz_score = get_quiz_score()
16 exam_score = get_exam_score()
17 average = compute_average(quiz_score, exam_score)
18 print("Status:", check_pass(average_score))
```

H3

Sales Tracker v3

Making a more robust console tracker

Sales Tracker v3

```
1 def add(total):
2     item_cost = int(input("Enter item cost: "))
3     item_count = int(input("Enter item count: "))
4     total_item_cost = item_cost * item_count
5     return total + total_item_cost
6 def sub(total):
7     """Remove total item cost (cost, count) from total and return"""
8 def show(total):
9     """Print total"""
10
11 def main():
12     total = 0
13     running = True
14     while running:
15         command = input("Provide command: ")
16         if command == "command 1":
17             total = add(total)
18         elif command == "exit":
19             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

```
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

```
1 try:
2     divider = int(input("Number: "))
3     budget = 1_000
4     print(budget / divider)
5 except ValueError:
6     print("Enter a valid number!")
7 except ZeroDivisionError:
8     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

```
1 def withdraw(balance, amount):  
2     if balance < amount:  
3         raise ValueError("Insufficient funds")  
4     if amount < 0:  
5         raise ValueError("Withdraw amount must be positive")  
6  
7     new_balance = balance - amount  
8     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

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

H4

Sales Tracker v4

Putting all of it together

Sales Tracker v4

```
1 def add(total):
2     """Add item cost (cost, count) from total and return"""
3 def sub(total):
4     """Remove item cost (cost, count) from total and return"""
5 def show(total):
6     """Print total"""
7
8 def main():
9     total = 0
10    running = True
11    while running:
12        command = input("Provide command: ")
13        if command == "command 1":
14            total = add(total)
15        elif command == "exit":
16            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
```



The image shows a top-down view of a person's hands typing on a laptop. The laptop screen displays a code editor with a dark theme. On the left, a file explorer shows a project structure with folders like 'src' and 'test', and files like 'App.js' and 'index.html'. The main editor area shows a JavaScript file named 'App.js' with code for a class component. The code includes a constructor, a render method, and a stateful method. A red rectangular box is overlaid on the screen, containing the text 'Positive Input' in white. The person's hands are positioned over the keyboard, with fingers pressing keys. The background is a plain, light-colored surface.

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 there was a draw:

You win!

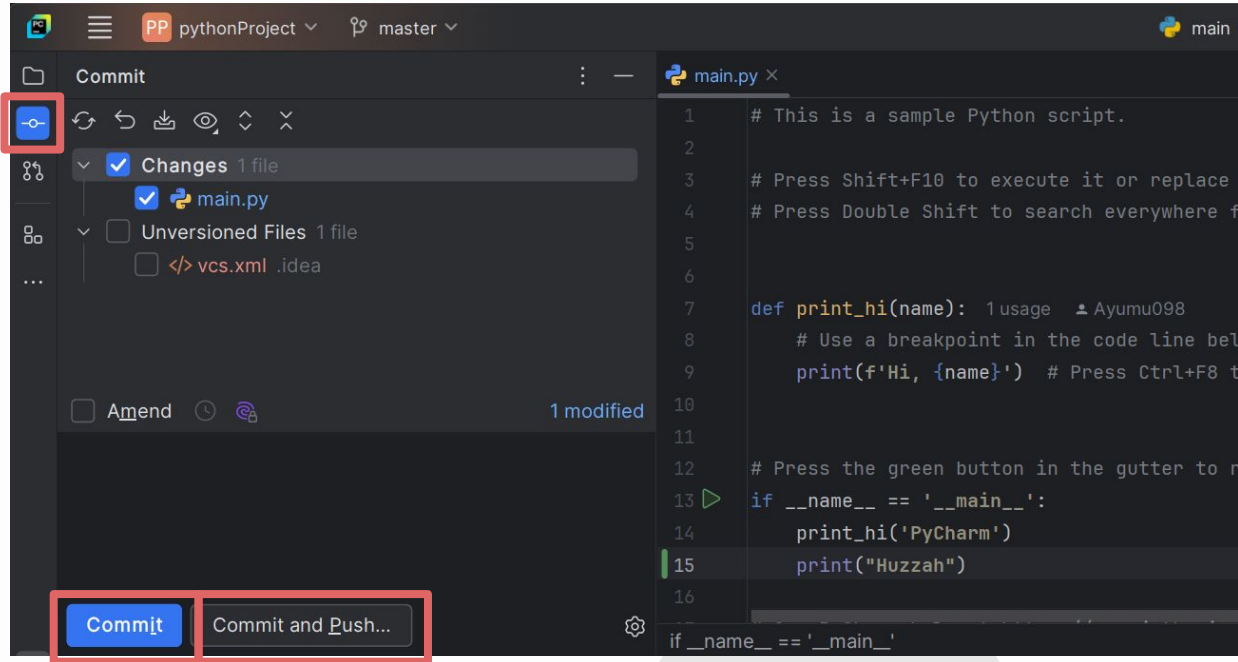
You Lost!

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

File Handling

Outside-Code Storage

05

Comprehension

Iteration Shortcut

06

Lab Session

Culminating Exercise

Python: Day 01

Introduction to Python and Basic Syntax