Python: Day 01

Introduction and Control Flow

Objectives

Foster a Strong Foundation in Python

Understand the fundamental programming concepts and how to use them properly

Develop Problem Solving Skills

Gain practical experience through exercises and lab sessions

Prepare for Specialization

Provide insights to how Python can be used in various industries

Agenda

01

Introduction

What is Python?

02

Variables

Data Storage

03

Control Flow

Processing Information

04

Functions

Grouping Control Flows

05

Error Handling

Handling invalid code

06

Lab Session

Culminating Exercise

01

Introduction

Overview of Python’s characteristics and potential

“Python is an **interpreted**, **object-oriented**, **high-level** programming language with **dynamic semantics**. ”

— python.org

Key Features

Modern

Constantly updated with new, useful features

Convenient

Simple and concise for easier development

Active

Large community with a rich ecosystem

“Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, Scheme, or Java. ”

— python.org

Java

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

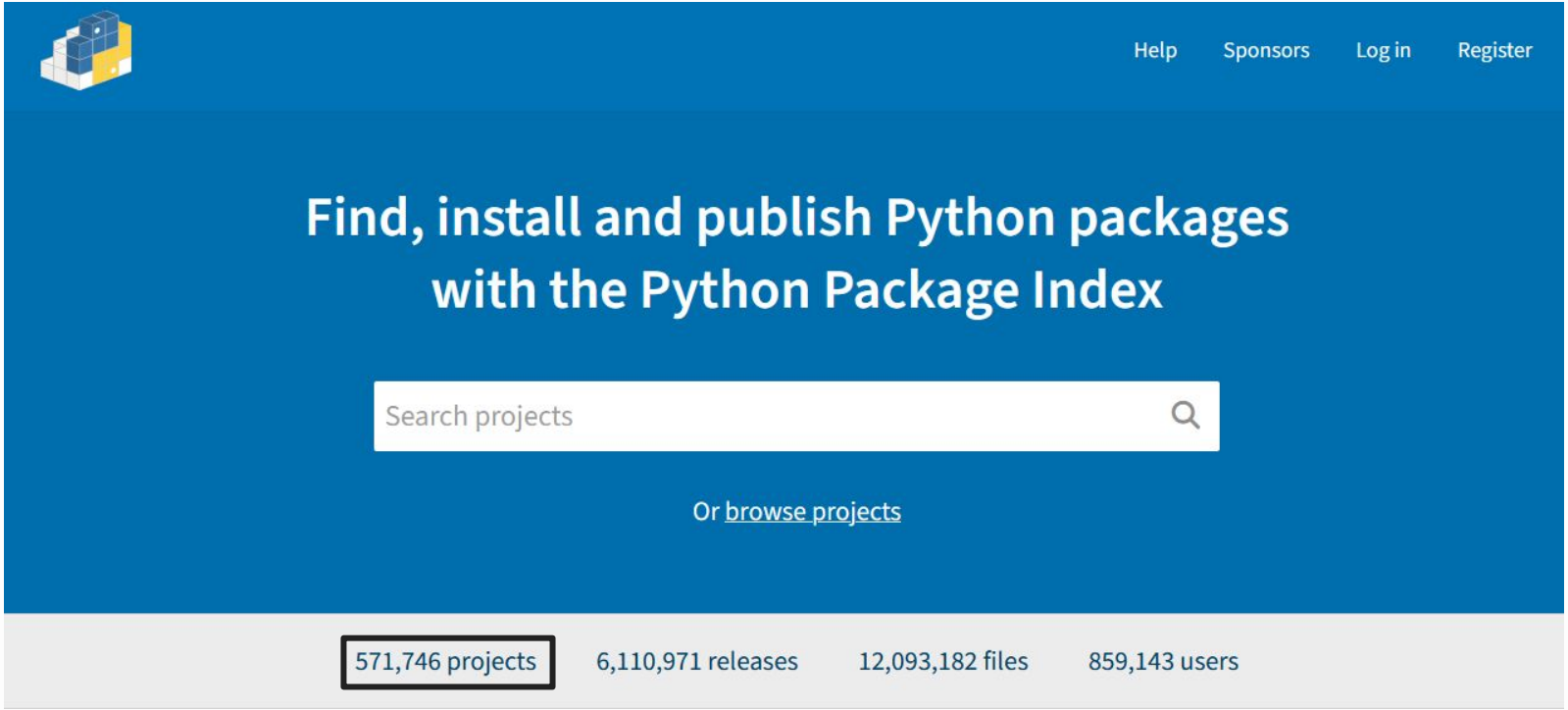
C++

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

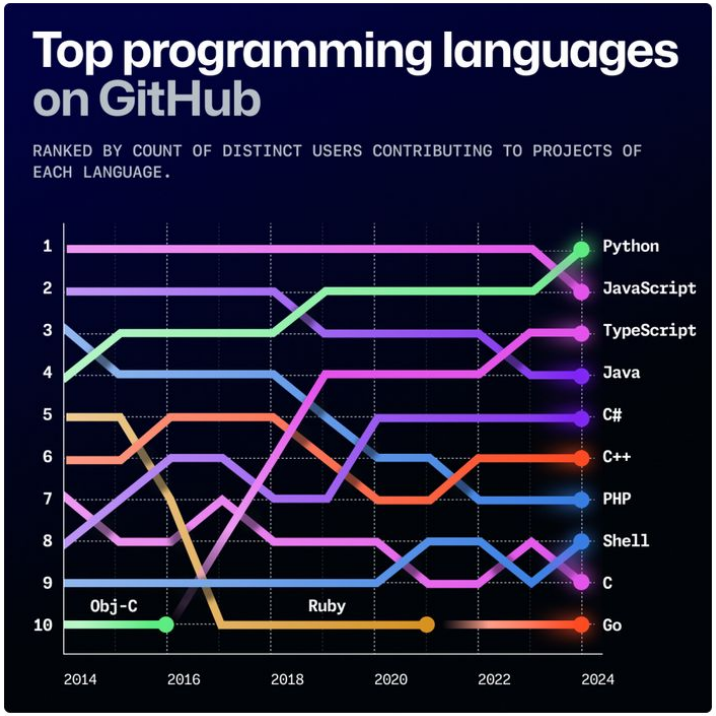
Python

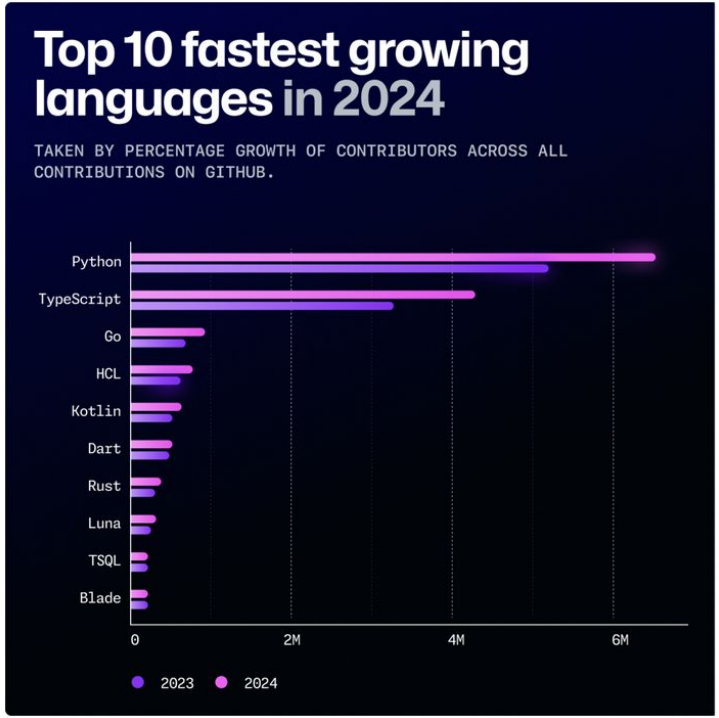
```
1 print(“Hello World”)
```

Python Packages (pypi.org)



Python Growth

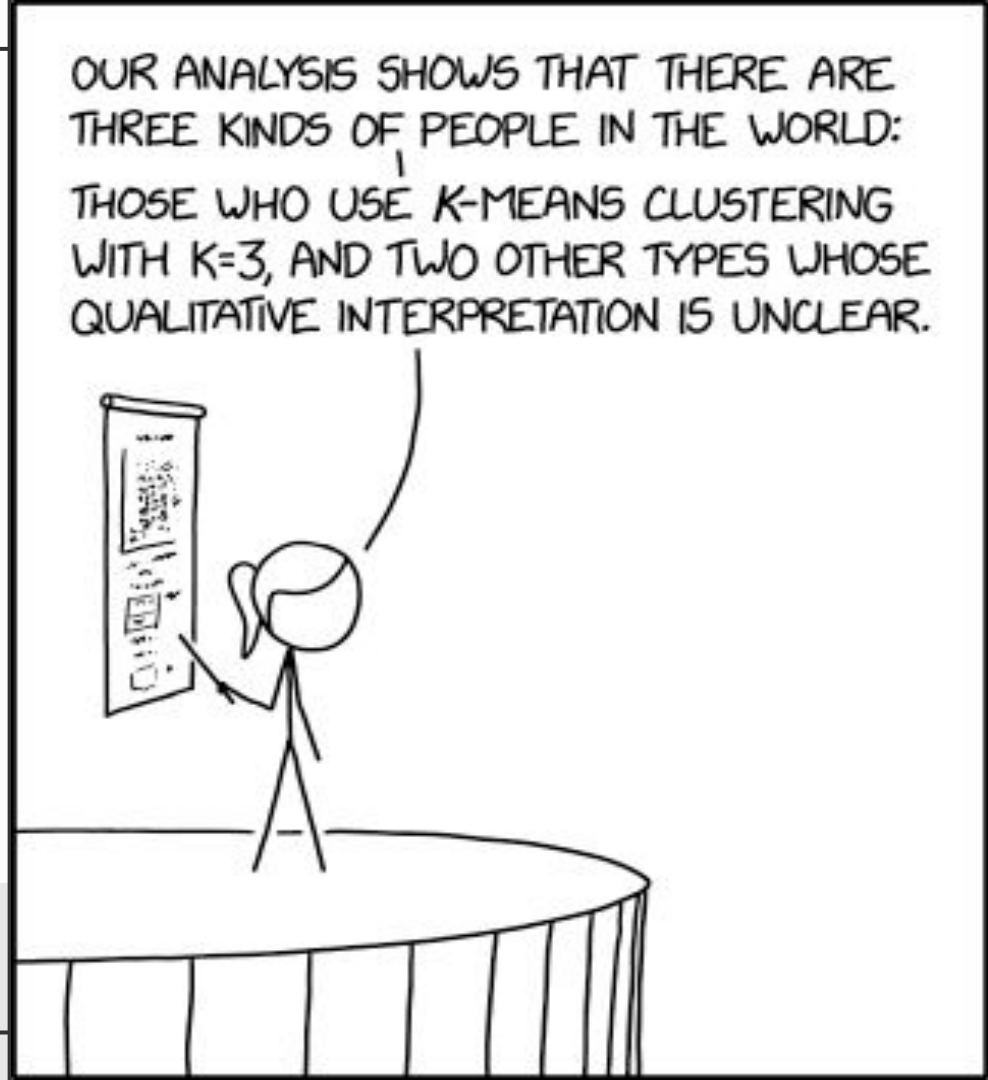




Where can you use Python?

Data Science

Data Science is a very versatile field that includes data processing, cleaning, visualization, and analysis.

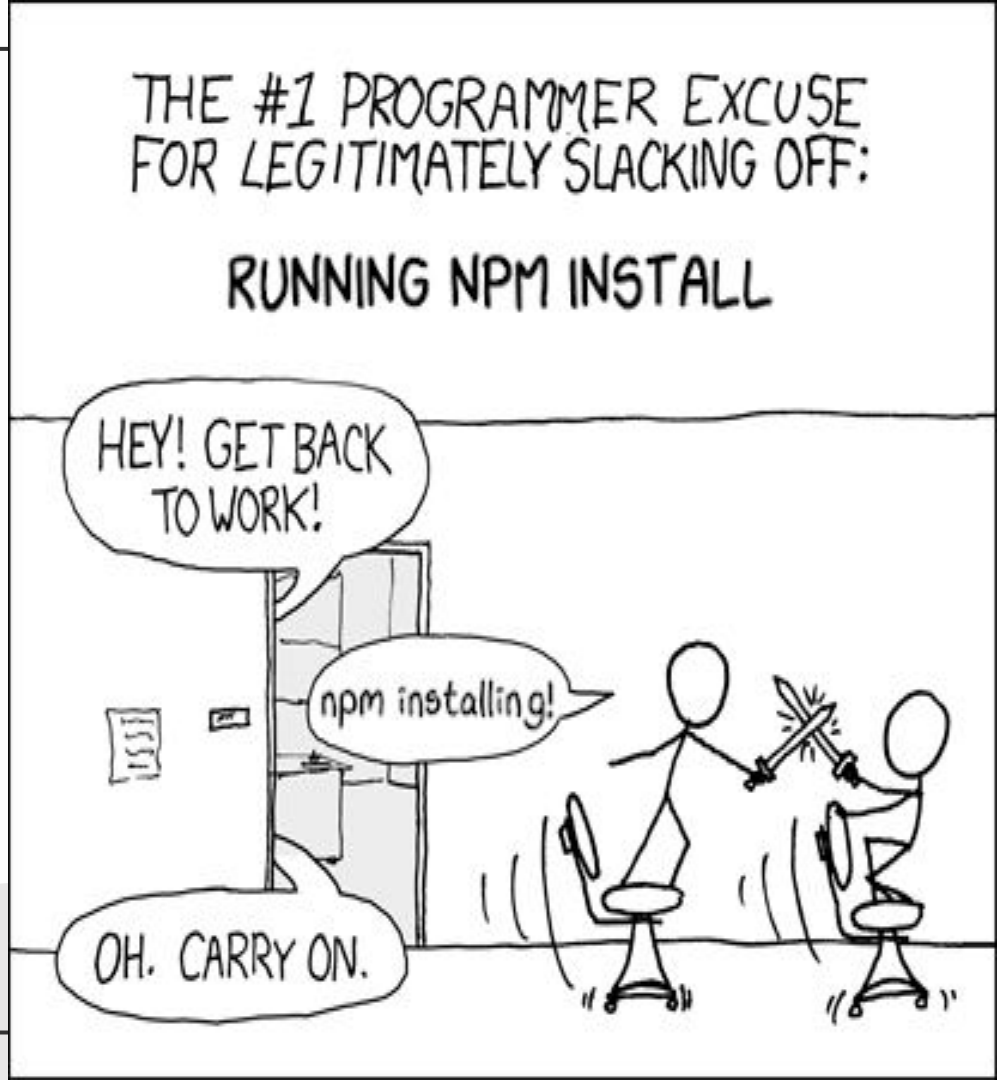


Machine Learning

Machine Learning is a specialized field for classifying, predicting, and analyzing quantitative data.

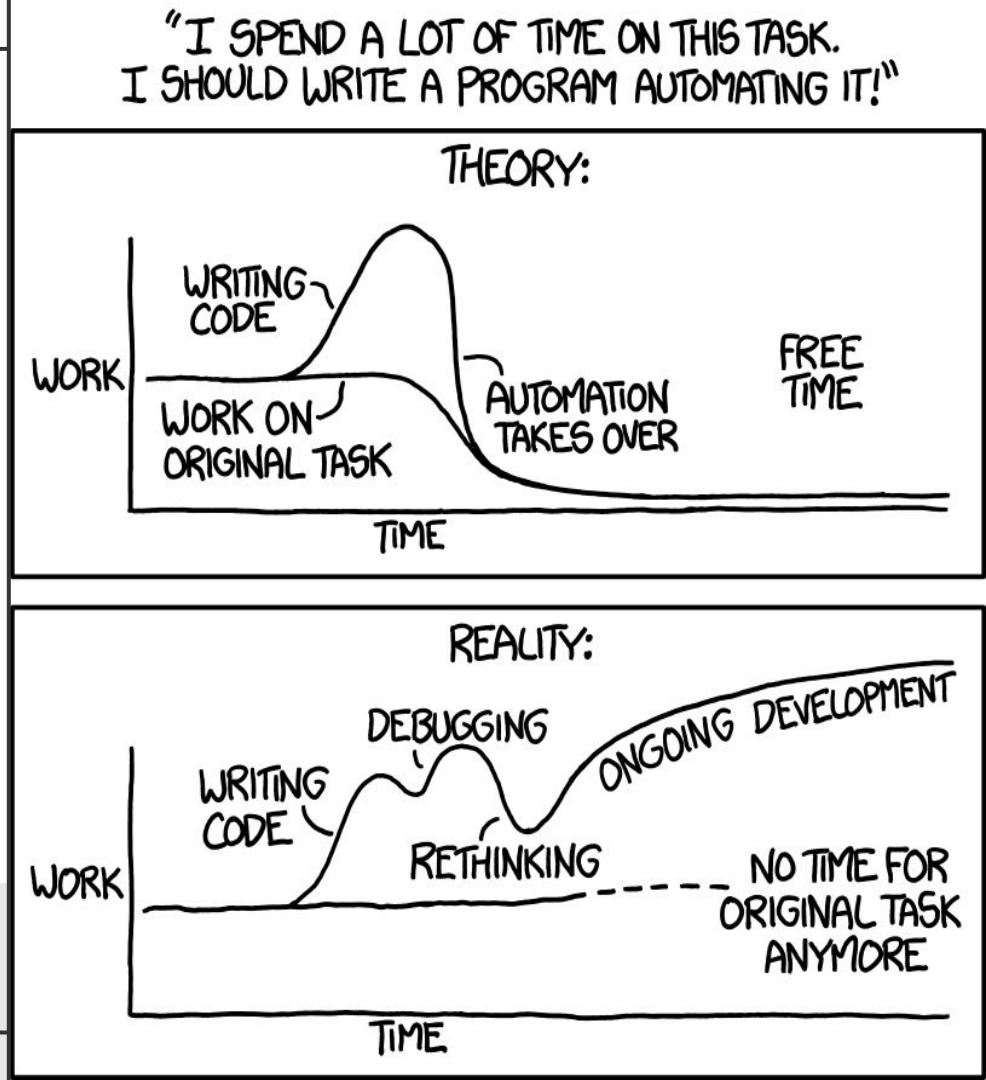
Web Development

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



Automation

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



Specialist Fields

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

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, with a guiding principle:
"There should be one— and preferably only one —obvious way to do it".

Setup Pycharm

Preparing our Integrated Development Environment (IDE)

Code Editor Support

Formatting

Ensures proper indentation, layout, and structure for readable code

Linting

Identifies syntax errors, unused components, typos, and potential bugs

Highlighting

Differentiate components with consistent colors for less cognitive load

Autocompletion

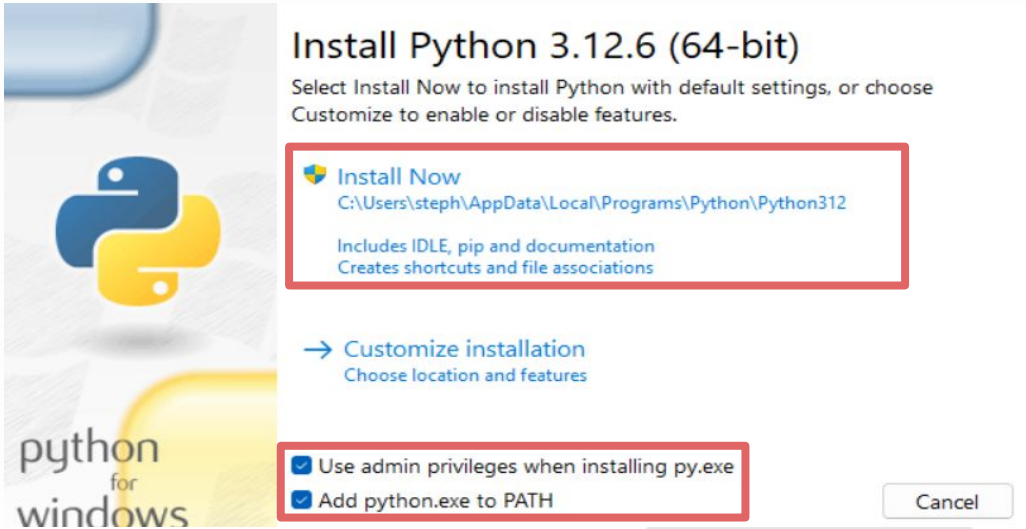
Autocomplete common code components or use templates

Step 1: Download Python Downloader

Go to https://www.python.org/downloads/ and click the first download button. The version and type will be automatically detected.

Step 2: Run Python Installer

- Run the downloaded installer (preferably with admin privileges)
- Enable all of the checkbox options
- Select Install Now option

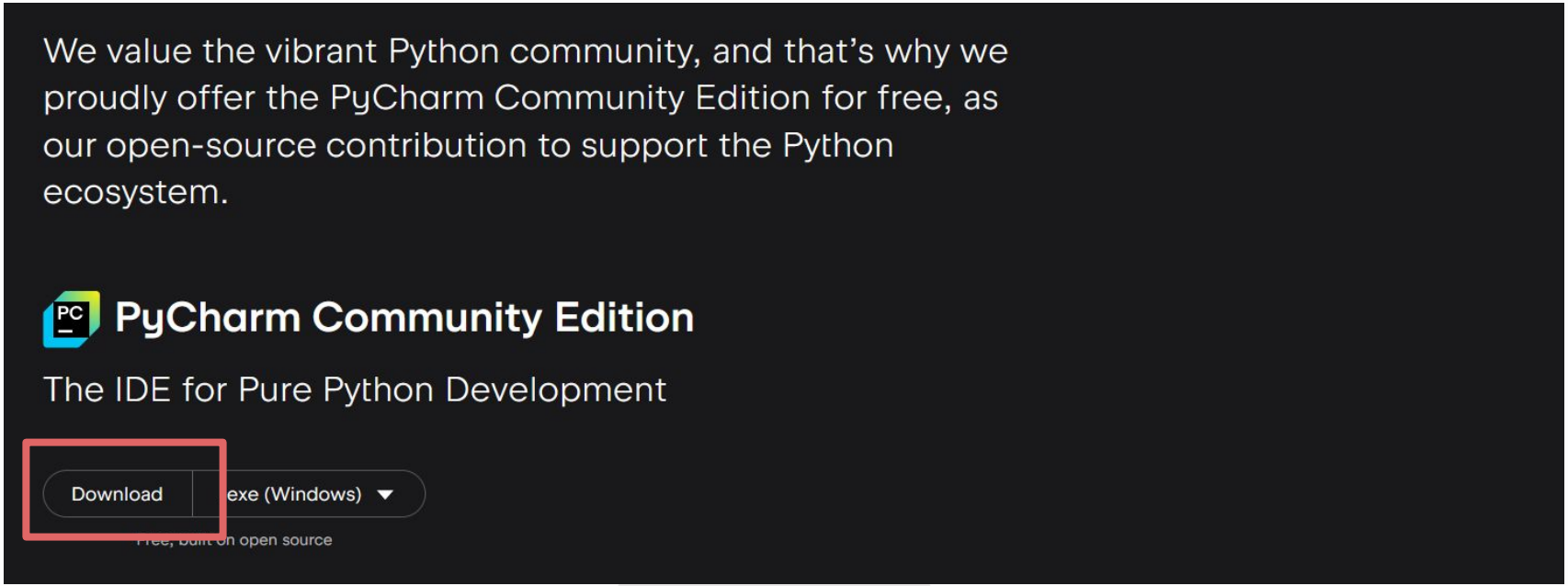


Step 3: Download PyCharm Installer

Go to https://www.jetbrains.com/pycharm/ and **scroll down** (**don't click first download option**)

Step 3: Download PyCharm Installer

Install the **PyCharm Community Edition** by clicking the **Download** option.

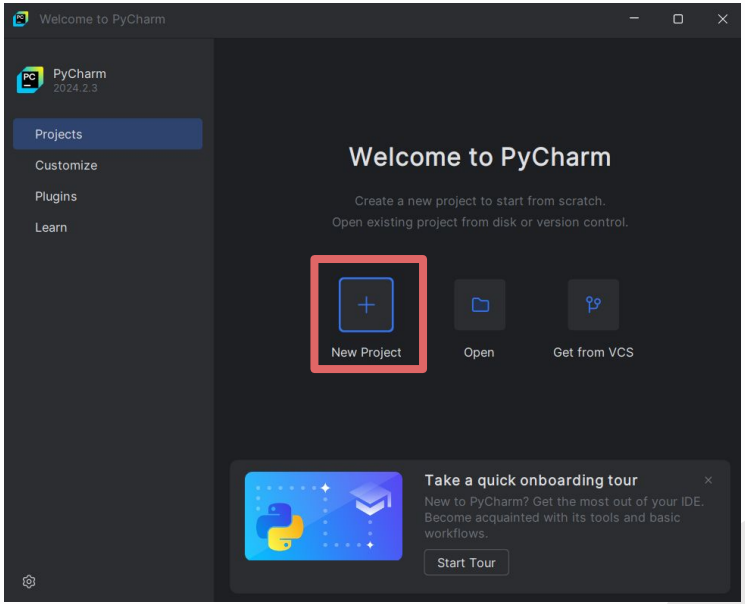


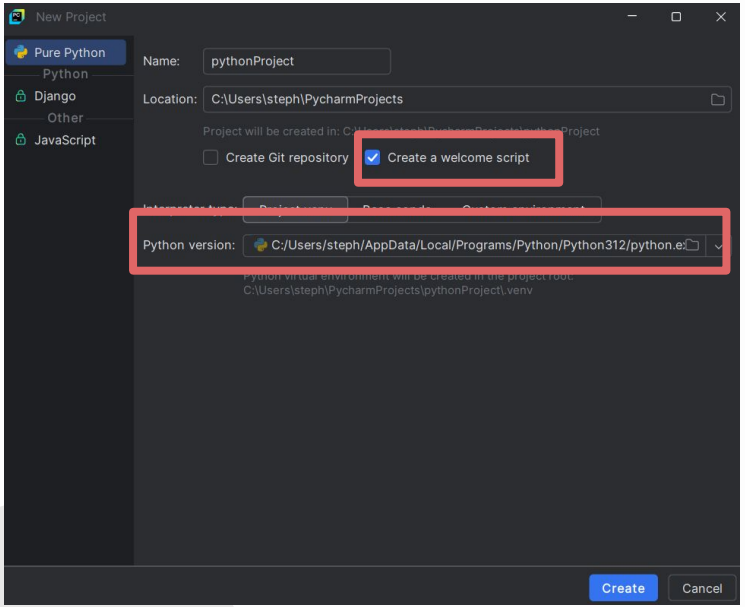
Step 4: Run PyCharm Installer

Run the PyCharm Installer (preferably with admin privileges) and use the default options.

Step 5: Configure Python Interpreter

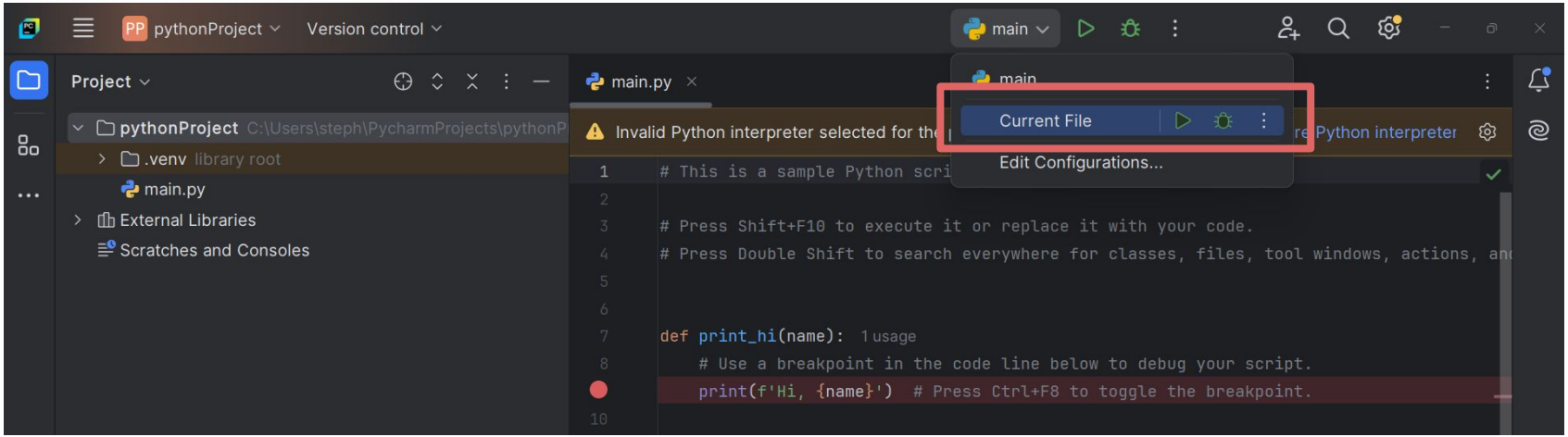
Click New Project, then select **“Create a welcome script”.**
Python version should be automatic. Leave it as it is.





Step 6: Run your First Code!

For convenience, select the dropdown with main and select **Current File**.
Finally, click the play button on the right or press `CTRL + SHIFT + F10` to run.

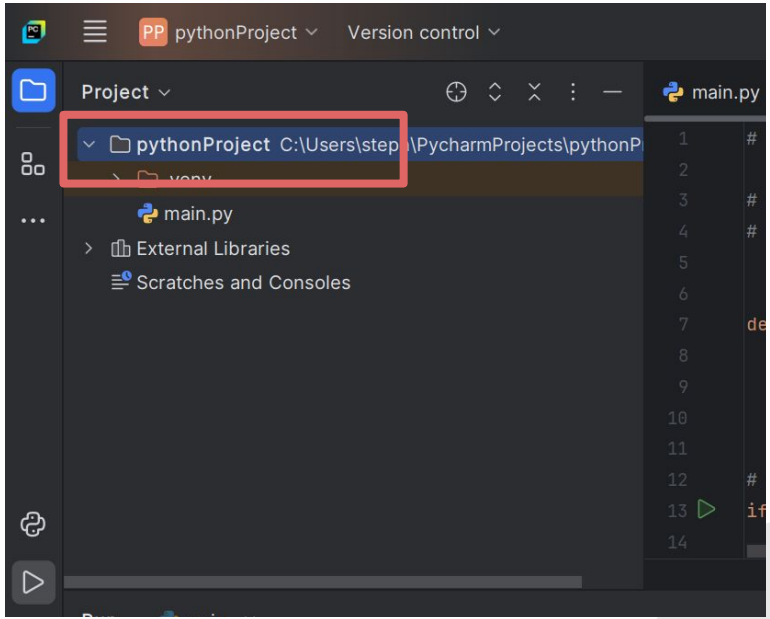


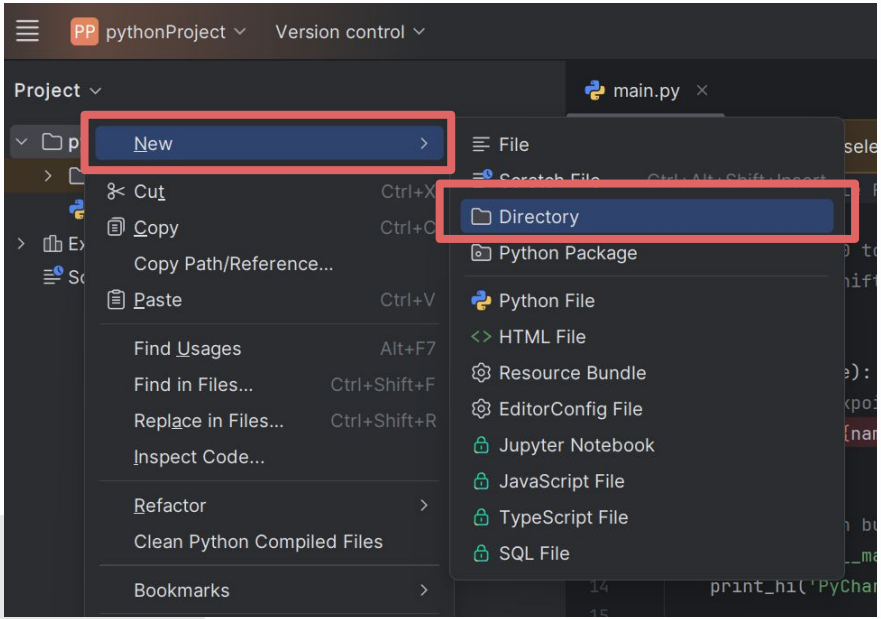
Hello World

A journey of a thousand miles begins with a single step

Step 1: Create a New Folder

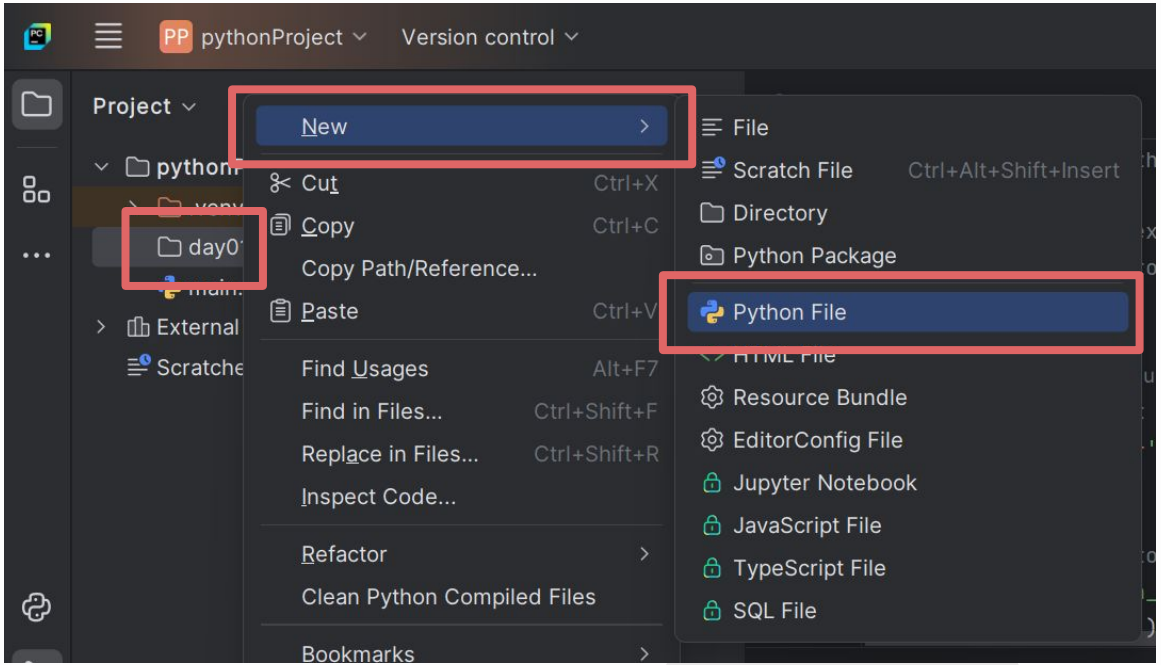
Right click the current project folder name, select **`New > Directory`**.
When prompted, name it **`day01`**





Step 2: Create New Python File

Right click the new folder name, select **`New > Python File`**. Name it **“hello”**



Building your First Code

print ()

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

Building your First Code

```
print(Hello World )
```

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

Building 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! ")
```

Print the following in the console

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

TIP: Make a new file

Single-Line Comment

If you want to write a line without being detected as code, use a hashtag

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

```
1  # print("Hello World")
2  print("Hello Again")
```

TIP: Try this with your current code

Multiple-Line Comment

If you want to write multiple lines without being detected as code, use triple quotes

```
1  """
2  print("Hello World")
3  print("Hello Again")
4  """
```

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!")
```

02

Variables

Representing and storing data in Python

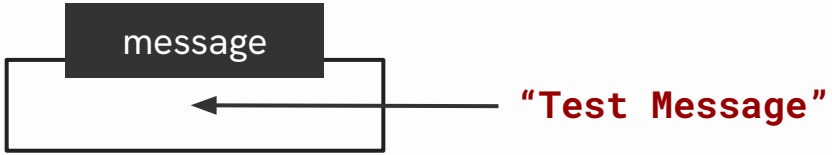
Variable Define

Creating your own data and storing them

Variable Declaration

Variables are declared or assigned using the following format

```
1  message = “Test Message”
```



Variable Printing

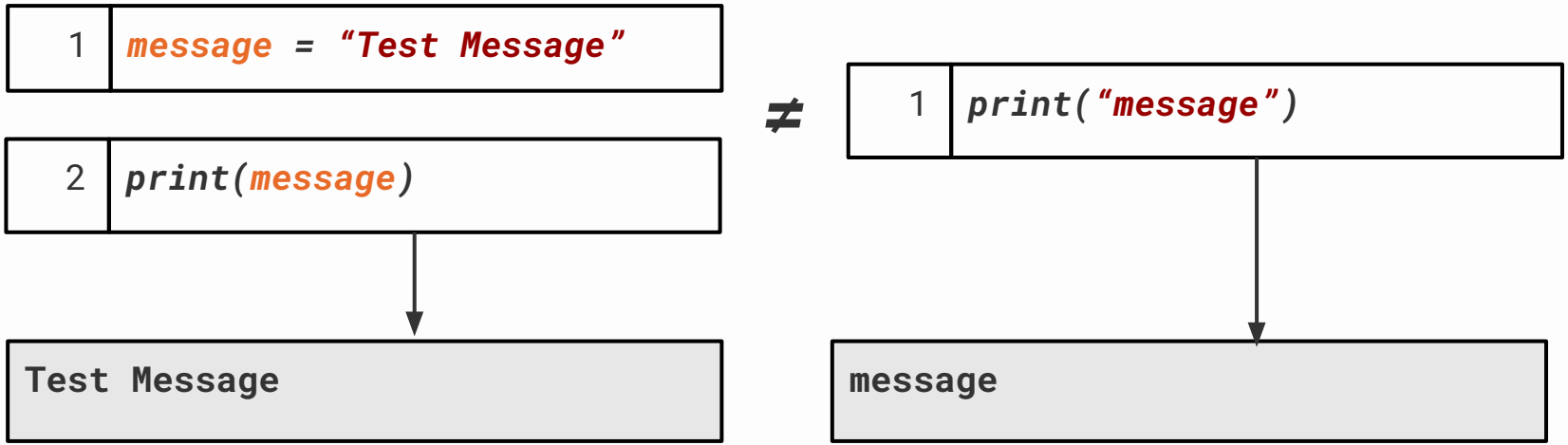
Variables can also be printed with the **`print`** function

```
1  message = “Test Message”
```

```
2  print(message)
```

Variables and Text

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



Variable as Nicknames

One of the key reasons to use variables is for representing data concisely.


```
2  print(name)
```

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

Change your Earlier Code

Old Code:

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

New Code:

```
message = "Hello! My name is your name."
extra_message = "I am learning Python!"

print(message)
print(extra_message)
```

Variable Change

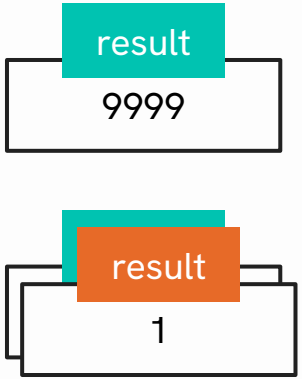
Updating existing variables

Variable Reassignment

Variables can be changed by using the same name again

```
result = 9999
print(result)

result = 1
print(result)
```

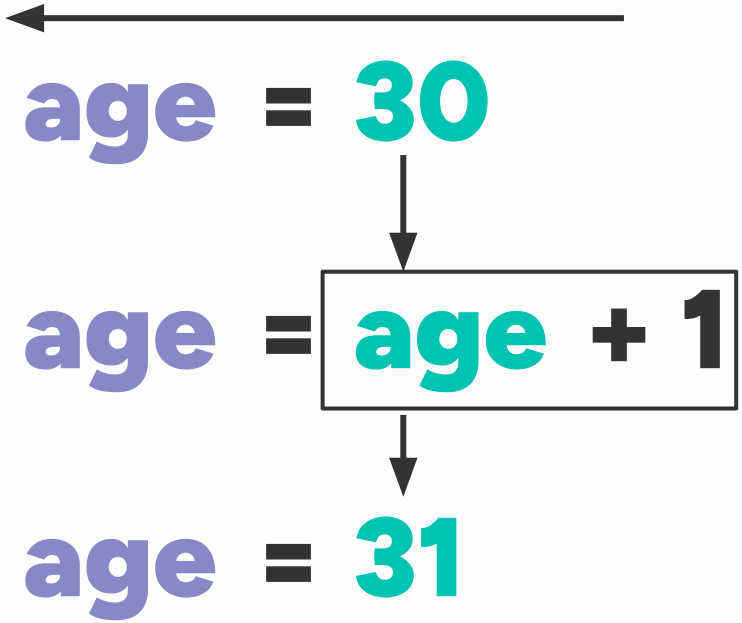
```
9999
1
```



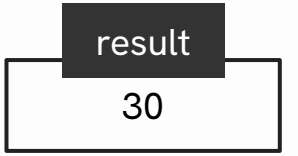
TIP: Try this code in a new file

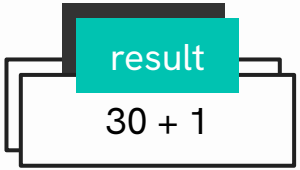
Variable Evaluation

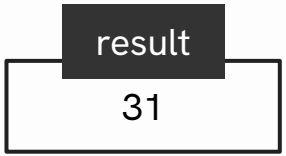
Python evaluates code top to bottom, right to left




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







Quick Exercise: Result Changing

Given the following variable:

```
result = 9999
```

Change `result` to `result` plus one and print it:

```
10000
```

Then, change the `result` again to `result` times three and print it:

```
30000
```

TIP: Make a new file for this

Variable Name

Learn the rules for naming variables

Variable Naming

- Variable names are case-sensitive and can use all letters of most alphabets
- Variable names can use numbers (0-9) but it can’t be at the start of the name
- Variable names cannot use special characters except underscores (_)

Incorrect Examples

```
1 3example = 123
```

```
2 variable spaced = “Siomai Sioman Shupao”
```

```
3 alert! = “This is important!”
```

Quick Exercise: Is this valid?

```
1  correct = “True”
```

```
2  years taken beforehand = 12
```

```
3  _hidden = “Please keep this a secret”
```

```
4  $var = 123
```

```
5  million_dollars = 1000000.00
```

```
6   何でもない  = ””
```

Variable Types

The basic forms of data available in Python

Strings (str)

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

```
1 empty_string_a = ''
```

```
2 empty_string_b = ""
```

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

Make a new file and try printing each variable!

Integers (int)

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

```
1 zero = 0
```

```
2 negative_number = -1
```

```
3 large_number = 111_222_333
```

Make a new file and try printing each variable!

Floating-Point Numbers (float)

Floats represent real positive, or negative numbers with decimal points.

```
1 zero_float = 0.0
```

```
2 negative_float = -1.2
```

```
3 long_float = 111_222.333_444
```

Make a new file and try printing each variable!

Boolean (bool)

Booleans represent True or False

```
1 is_raining = True
```

```
2 has_red_hair = False
```

Make a new file and try printing each variable!

None (None)

None represent null or empty values

```
1 example = None
```

Make a new file and try printing each variable!

Quick Exercise: Favorite Sample

Create the following variables and provide the appropriate information:

```
favorite_food_name = Your favorite food's name
favorite_food_price = How much does it usually cost to make it?
is_homemade =  Is it a type of food that's homemade?
```

Then, print each information one line at a time

TIP: Make a new file for this

Type Function

To determine the data type of a direct data or variable, you can use the `type()` function. This outputs a string to show the data type of a variable.

```
1  type("Hello World")
```

To actually get the type of a data, you can either place the result in a variable and print

```
1  data_type = type("Hello World")
2  print(data_type)
```

Quick Exercise: Print the Data Type

```
1  cause = “It’s been raining in Manila”
```

```
2  cold = False
```

```
3  hard = 1
```

TIP: Make a new file for this

Print Parameters

Extra features for printing data in the console

Multiple In the Same Line

Multiple inputs can be provided by using **commas to separate them**.
Python will print them in the same line and add spaces in-between.

```
1  print("First Message", "Second Message", "Third Message")
```

```
1  first = "First Message"
2  second = "Second Message"
3  message = "Third Message"
4
5  print(first, second, message)
```

Make a new file and try either options!

Quick Exercise: Print the Following

Define the following variables

```
user_name = Your name here
favorite_number = favorite number
favorite_color = favorite color
```

Then print the following in the console

```
Hello, my name is: Your name here
My favorite number is: favorite number
My favorite color is: favorite color
```

TIP: Make a new file for this

Print Separator Parameter

The `print()` function can also accept additional, special inputs.
One of these is `sep` that sets the separator between multiple inputs.

```
1  print(1, 2, 3, 4, sep=" | ")
```

```
1 | 2 | 3 | 4
```

```
2  print(1, 2, 3, 4, sep="")
```

```
1234
```

Make a new file and try either!

Quick Exercise: Print the Following

Generate the following results

```
1 -> 2 -> 3 -> 5 -> 6
```

```
1 + 2 + 3 + 5 + 6
```

TIP: Make a new file for this

Print End Parameter

The `end` parameter sets the final character added (the default is newline `"\n"`)

```
1  print("First", end="! ")
2  print("Second", end="! ")
```

```
First! Second!
```

Make a new file and try this!

Escape Characters

To add special characters in Python strings, use an escape key `\`

Character	Symbol
Single Quote	`\”`
Double Quote	`\’`
Backslash	`\\`
Tab	`\t`
Newline	`\n`

Quick Exercise: Print the Following

Given the following variables

```
first = “First Message”
second = “Second Message”
```

Create the following outputs (without editing the variables):

```
First Message!!!
Second Message...
```

TIP: Make a new file for this

Challenge: Create the Following

Given the following variable:

```
message= “Paulit-ulit na lang”
```

Print the following message (pay attention to the punctuations)

```
Paulit-ulit na lang. Paulit-ulit na lang. Paulit-ulit na lang!
Paulit-ulit na lang. Paulit-ulit na lang. Paulit-ulit na lang!
Paulit-ulit na lang. Paulit-ulit na lang. Paulit-ulit na lang!
```

TIP: Make a new file for this

Input Function

Using data given by the user of the code

Input Function

To take input from users, you can use the **`input`** function. This function creates an output that you can store in a variable. Note: The input will always return a str.

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

Make a new file and try this!

Input Function (With Text)

To take input from users, you can use the **input** function. This function creates an output that you can store in a variable. Note: The input will always return a str.

```
1  user_input = input("Type anything in the console: ")
2  print(user_input)
```

Edit the earlier code!

Return Values

The reason we need a variable to is so we can use the result later. Yes, it does the process, but it needs to return a result that we can use?

```
1  input("Type anything in the console: ")
2  print()
```

```
1  user_input = input("Type anything in the console: ")
2  print(user_input)
```

Quick Exercise: Ask Input and Print

Ask the user for the following inputs

```
user_name = Your name here
favorite_number = favorite number
favorite_color = favorite color
```

Then print the following in the console

```
Hello, my name is: Your name here
My favorite number is: favorite number
My favorite color is: favorite color
```

TIP: Make a new file for this

Type Conversion

Making operations and processes compatible with Python

User Input Issue

What is the problem with this code?

```
number_input = input(“Type any number in the console: ”)
print(number_input + 1)
```

Make a new file and try this

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 int with the `int()` function. Try the following:

```
number_input = input("Type any number in the console: ")
print(number_input + 1)
```

```
number_input = int(input("Type any number in the console: "))
print(number_input + 1)
```

Quick Exercise: Number Adder

Ask the user for the following inputs

```
first_number = First Number
second_number = Second Number
```

Then print the following in the console

```
Sum of Two Numbers
```

TIP: Make a new file for this

Integer Type Conversion

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

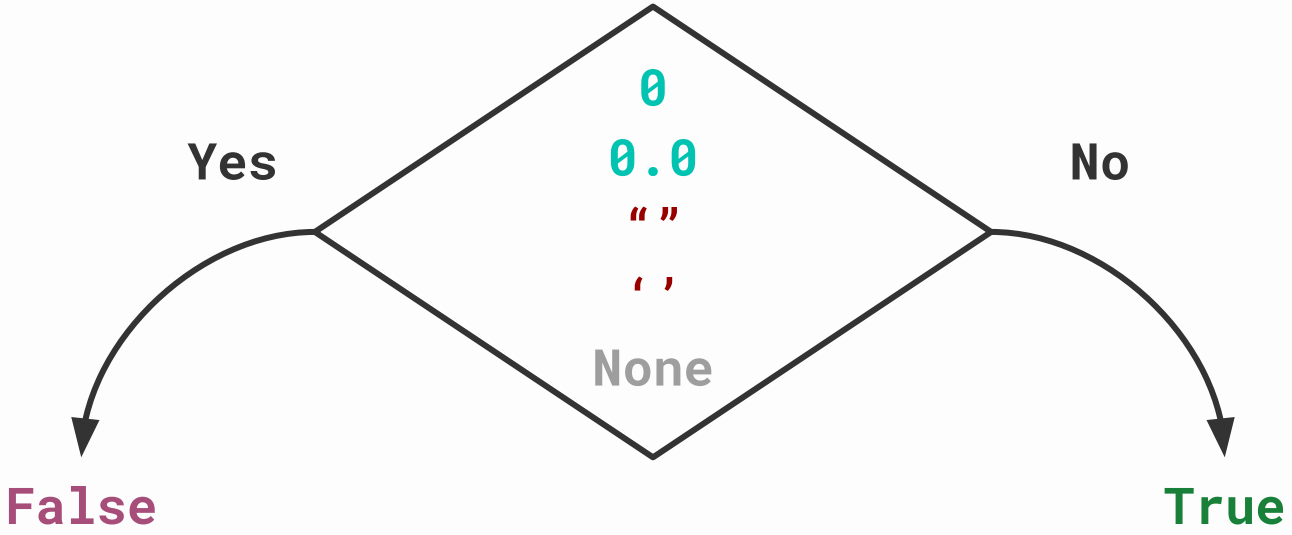
Original Data Type	Behavior
Float	Drops all decimal places
Boolean	True →1, False → 0
String	Converts to integer. If invalid, raises an error
None	Raises an error

Float Type Conversion

You can convert most basic data types to float with the `float()` function.

Original Data Type	Behavior
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

Boolean Type Conversion



Boolean Type - Common Pitfalls

Here are common pitfalls when converting to Boolean with the `bool()` function

Value	Notes
`" "`	Spaces are not completely empty → **`True`**
`' '`	Spaces are not completely empty → **`True`**
`"False"`	Non-empty string → **`True`**
`"None"`	Non-empty string → **`True`**

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
//	Floor Division	`result = 11 // 2`	5
**	Exponent/Power	`result = 11 ** 2`	121
%	Modulo/Remainder	`result = 11 % 2`	1

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 by using the multiplication operator.

```
1 print(“ice “ * 3)
```

```
ice ice ice
```

Update Shortcut

All operations where the variable is changed by a copy of it can be simplified.
Here are some examples:

Original Statement	Shortcut
`result = result + 5`	`result += 5`
`result = result * 10`	`result *= 10`
`message = message + “World”`	`message += “World”`

Quick Exercise: Counter Logic

Given the following variable:

```
counter = 0
```

Change the result to **result plus one** and print it:

```
print(counter)
```

Change the result to **result minus one** and print it:

```
print(counter)
```

TIP: Make a new file for this

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)
```

TIP: Make a new file and try this

PEMDAS

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)
```

Add this to the earlier code!

Quick Exercise: Body Mass Index

Ask the user for the following inputs

```
weight = Weight in Kilograms
height_foot = Height (foot part)
height_inches = Height (inches part)
```

Then print the BMI of the given input

TIP: Make a new file for this

$$\text{BMI} = \frac{\text{Weight (kg)}}{((\text{Feet} \times 12 + \text{Inches}) \times 0.0254)^2}$$

String Formats

Combine strings and variables conveniently

String Placeholder

You can put placeholders in strings in the form of curly brackets

```
1  message = “Hello {}! Nice to meet you!”
```

Then, you can replace the placeholders using the `.format()` syntax

```
2  name = input(“Enter your name: ”)
3  formatted_message = message.format(name)
4  print(formatted_message)
```

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

Make a new file and try this!

Quick Exercise: Scrapbook Sample

Create message templates and place them in variables

```
name_message = “Your name is {}.”
number_message = “Your favorite number is {}.”
color_message = “Your favorite color is {}.”
```

Then ask the user for their inputs

```
user_name = Your name here
favorite_number = favorite number
favorite_color = favorite color
```

Finally, print the templates with the placeholders replaced.

TIP: Make a new file for this

Multiple String Formatting

The format method supports multiple inputs as well as needed.

```
1  message = “Hello {} / {}”
2
3  name = input(“Enter your name: ”)
4  nickname = input(“Enter your nickname: ”)
5
6  print(message.format(name, nickname))
```

Make a new file and try this

Quick Exercise: Formatted Addition

Given the following inputs

```
number1 = int(input("First number: "))
number2  = int(input("Second number: "))
```

Print the following result

```
Addition: First number + Second number = Sum
```

TIP: Make a new file for this

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!")
```

Make a new file for each and try these

Quick Exercise: Scrapbook Sample

Given the following inputs

```
user_name = Your name here
favorite_number = favorite number
favorite_color = favorite color
```

Print the following (use f-strings)

```
Hello, my name is: Your name here. That's a nice name!
My favorite number is: favorite number. Nice choice.
My favorite color is: favorite color. Hopefully we see it today.
```

TIP: Make a new file for this

H1

Cost Calculator

Quick Practice of all the concepts discussed so far

Cost Calculator

Ask the user for the information of three items

```
item_1_price = Input your item price
item_2_price = Input your item price
item_3_price = Input your item price
```

Then print the final cost of the items

```
Total Cost: Total cost
```

Bonus Challenge:

Item Count?

03

Control Flow

Providing logic to data processing


PASS
FAIL

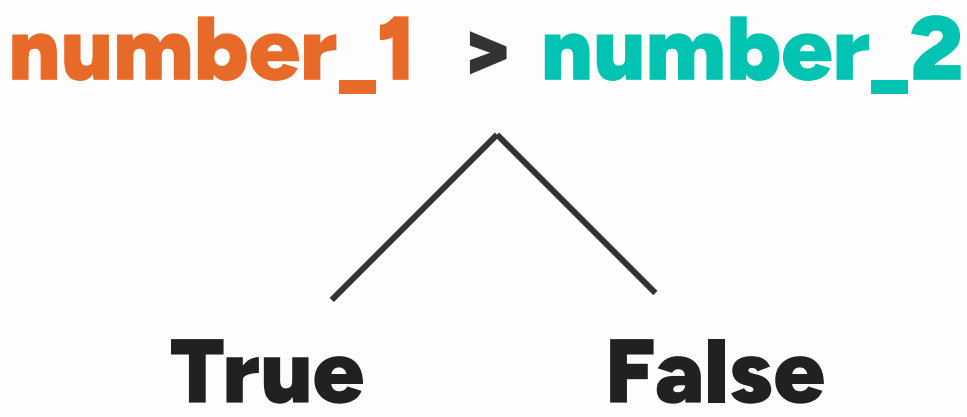
Relations

Checking if two values are related to each other

What are the possible results?

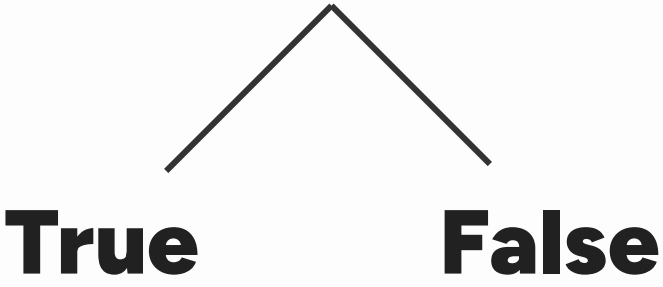
number_1 > number_2

What are the possible results?



You can also store the result

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`

Quick Exercise: Number Relations

Ask the user for the following inputs

```
first_number = First Number
second_number = Second Number
```

Then print the following in the console (use same-line printing)

```
First Number > Second Number
First Number < Second Number
```

TIP: Make a new file for this

Chained Relational Operator

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

Example
`3 < 11 < 20`
`3 <= 11 <= 20`
`20 > 11 > 3`
`20 >= 11 >= 3`

Quick Exercise: Number Range

Ask the user for the following inputs

```
min_number = First Number
max_number = Second Number
number = Third Number
```

Then print if the number is within min_number and max_number

```
True or False
```

TIP: Make a new file for this

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”`

Conditionals

Control when code executes

If Statement

The if statement allows you to run parts of the code when the given condition is **True**.
Note that anything placed as a condition uses the `bool()` function

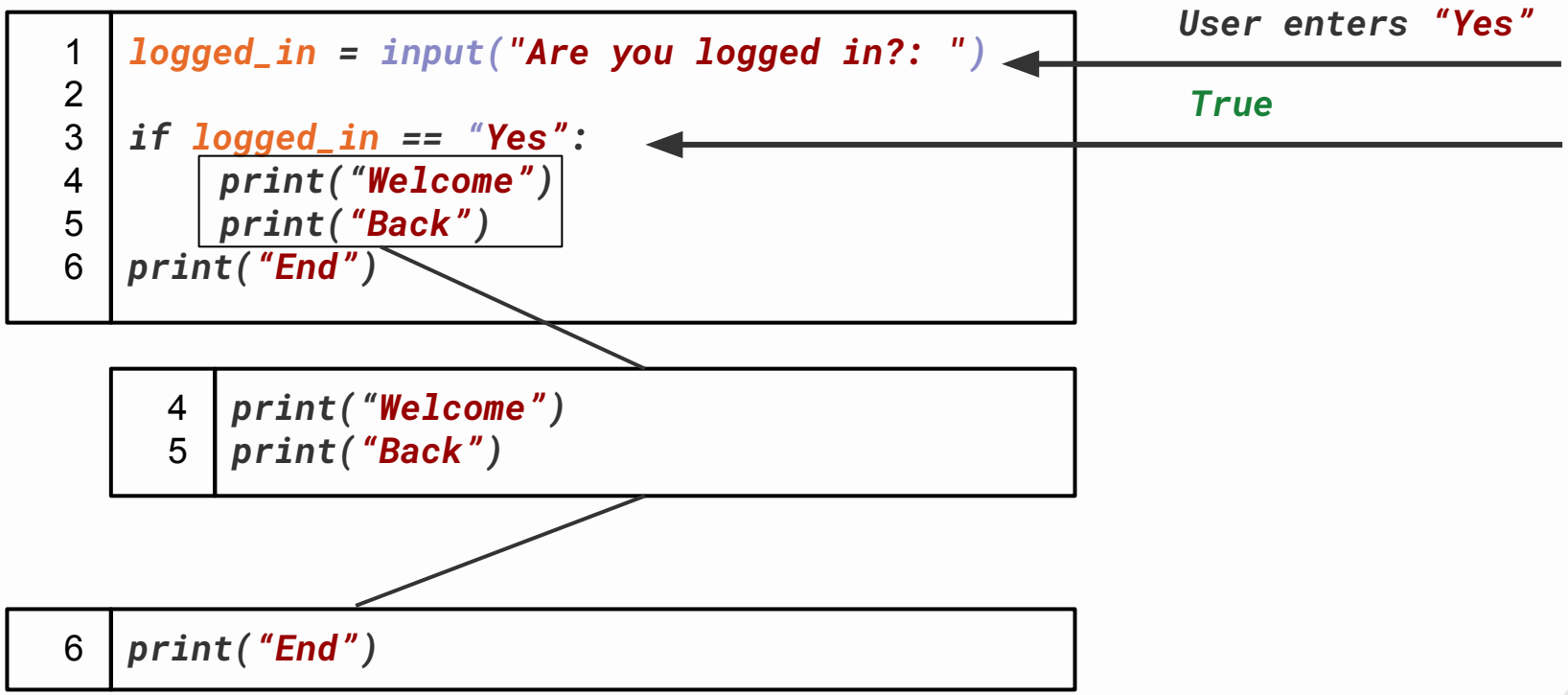
```
1  if condition:
2      # Processes here
```

```
1  logged_in = input("Are you logged in?: ")
2
3  if logged_in == "Yes":
4      print("Welcome")
5      print("Back")
6  print("End")
```

If Statement - True

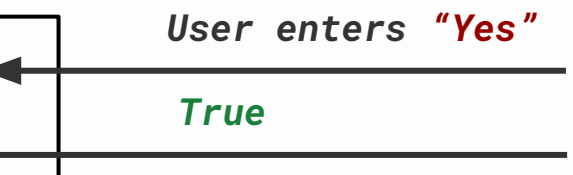
```
1  logged_in = input("Are you logged in?: ")
2
3  if logged_in == "Yes":
4      print("Welcome")
5      print("Back")
6  print("End")
```

If Statement - True



If Statement - True

```
1  logged_in = input("Are you logged in?: ")
2
3  if logged_in == "Yes":
4      print("Welcome")
5      print("Back")
6  print("End")
```




```
Welcome
Back
End
```

If Statement - False

```
1  logged_in = input("Are you logged in?: ")
2
3  if logged_in == “Yes”:
4      print(“Welcome”)
5      print(“Back”)
6  print(“End”)
```

User enters “No”

False

```
End
```

If Statement Example 02

```
1  number = int(input("Enter number: "))
2
3  if number > 0:
4      print("Number greater than zero!")
```

User enters "10"

True

```
Number greater than zero!
```

```
1  number = int(input("Enter number: "))
2
3  if number > 0:
4      print("Number greater than zero!")
```

User enters "-1"

False

```

```

Quick Exercise: Password Check

Create a variable with the password you want

```
correct_password = “pass”
```

Then ask the user for an input

```
password_input = input(“Please provide password: “)
```

Then print the following depending if the `password_input` equals `correct_password`

```
Access Granted!
```

TIP: Make a new file for this

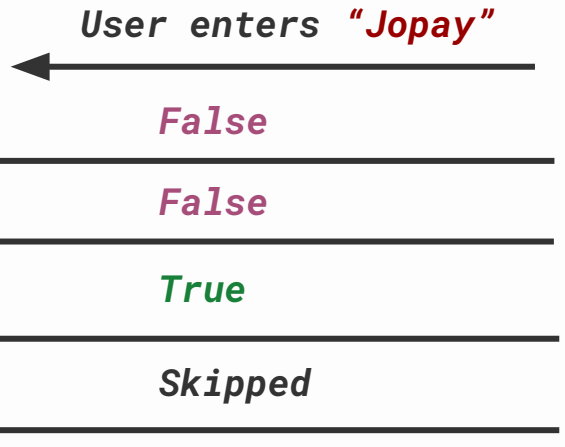
Elif Statement

The else-if or elif statement allows you to run parts of the code when the first condition is False but you want to do something else for other conditions

```
1  if condition_1:
2      # First Process
3  elif condition_2:
4      # Alternative Process No.1
5  elif condition_3:
6      # Alternative Process No.2
7  elif condition_4:
8      # Alternative Process No.3
```

Elif Statement Example

```
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")
```

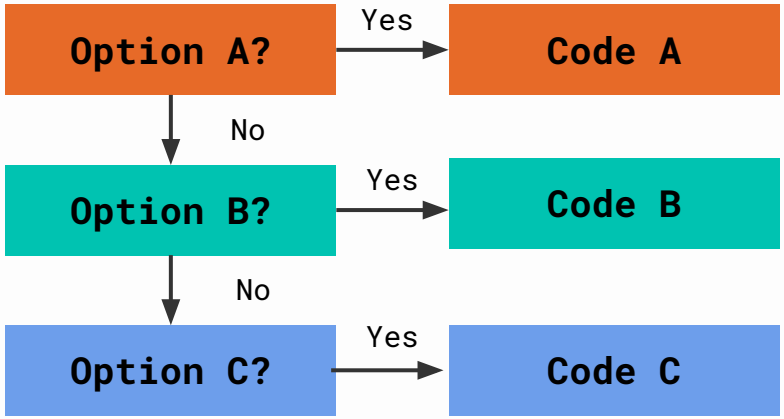



```
...kamusta ka na
```

Make a new file and try this code

Visualization Conditionals

Conditionals can be thought of as options for selecting what code to run



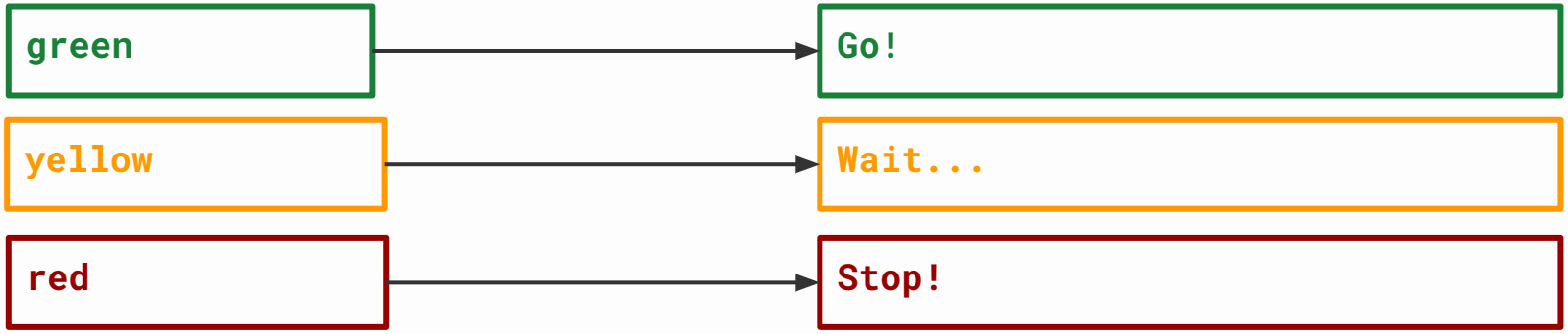

```
if condition_1:
    # Code A

elif condition_2:
    # Code B

elif condition_3:
    # Code C
```

Quick Exercise: Traffic Lights

Ask the user input for a color

```
color_input = input(“Please enter a color: “)
```

Then print the following depending on the color input



TIP: Make a new file for this

Else Statement (Last Resort)

The else statement allows you to run a part of the code when all else fails

```
1  if condition_1:
2      # Option A
3  elif condition_2:
4      # Option B
5  elif condition_3:
6      # Option C
7  else:
8      # Last Resort
```

Else Statement

The else statement allows you to run a part of the code when all else fails

```
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!")
```

Else Statement Example

```
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!")
```

User enters "Hey"

False

False

False

False

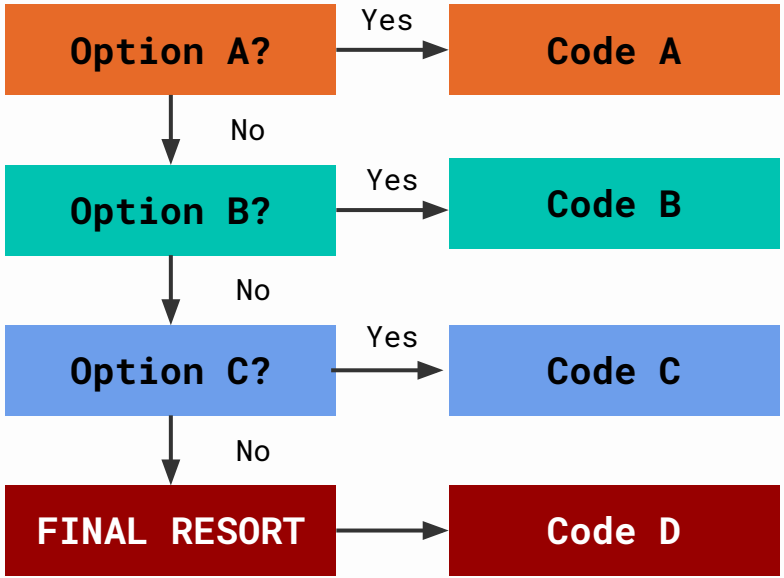
FINAL RESORT

```
I don't know that song!
```

Make a new file and try this code

Visualization Conditionals

Conditionals can be thought of as options for selecting what code to run



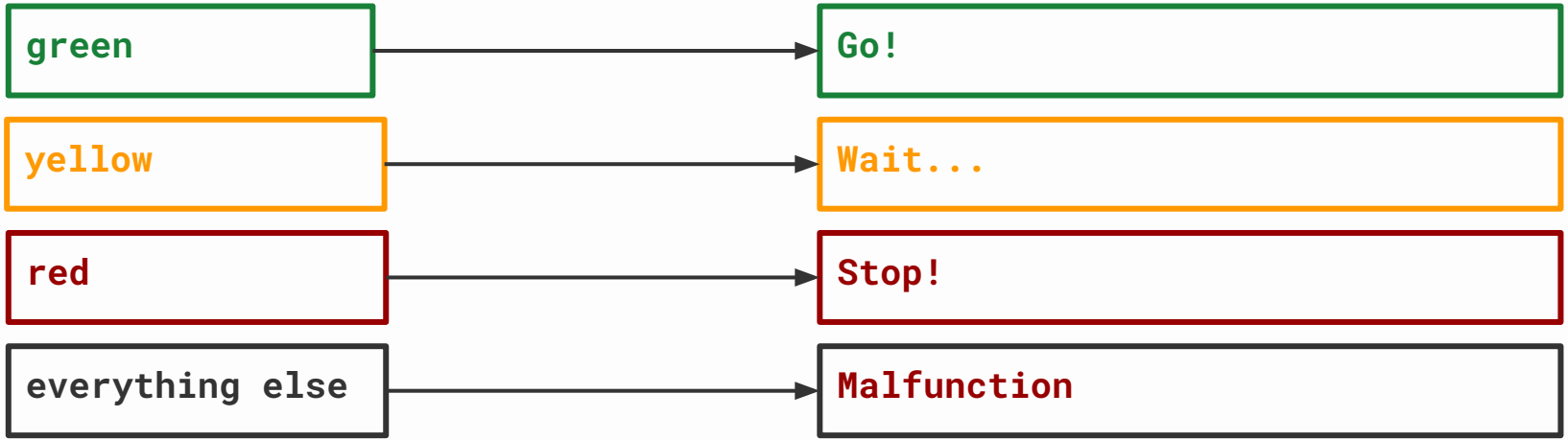

```
1  if condition_1:
2      # Code A
3  elif condition_2:
4      # Code B
5  elif condition_3:
6      # Code C
7  else:
8      # Code D
```

Quick Exercise: Traffic Lights (Complete)

Ask the user input for a color

```
color_input = input(“Please a color “)
```

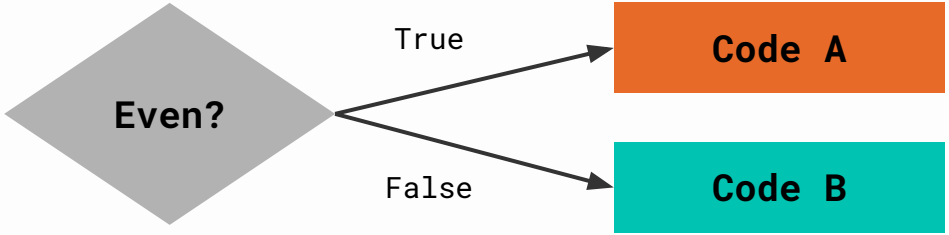
Then print the following depending if the color input



If-Else Condition

For most cases, only the if-else statements are used

```
1  number = int(input("Enter a number: "))
2  if number % 2 == 0:
3      print(f"Number {number} is even")
4  else:
5      print(f"Number {number} is odd")
```



Make a new file and try this code

Quick Exercise: Password Check (Update)

Create a variable with the password you want

```
correct_password = “pass”
```

Then ask the user for an input

```
password_input = input(“Please provide password: “)
```

Then print the following depending if the `password_input` equals `correct_password`

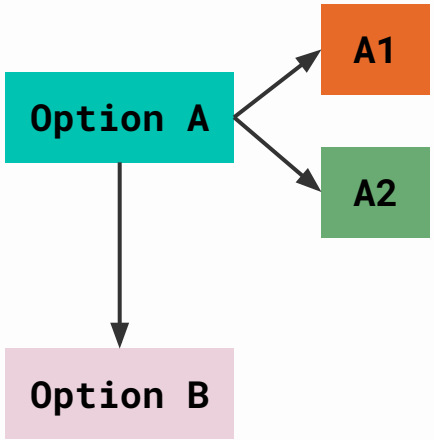
```
Access Granted!
```

```
Access Denied!
```

TIP: Make a new file for this

Nested If-Conditions

There is no limit on what code can be added inside a conditional - even another conditional!




```
1  age = int(input("Enter age: "))
2  is_member = input("Member? (y/n) ")
3
4  if age > 18:
5      if is_member == "y":
6          print("Welcome! You get discount")
7      else:
8          print("Welcome!")
9  else:
10     print("Not allowed!")
```

Make a new file and try this code

Nested If-Conditions

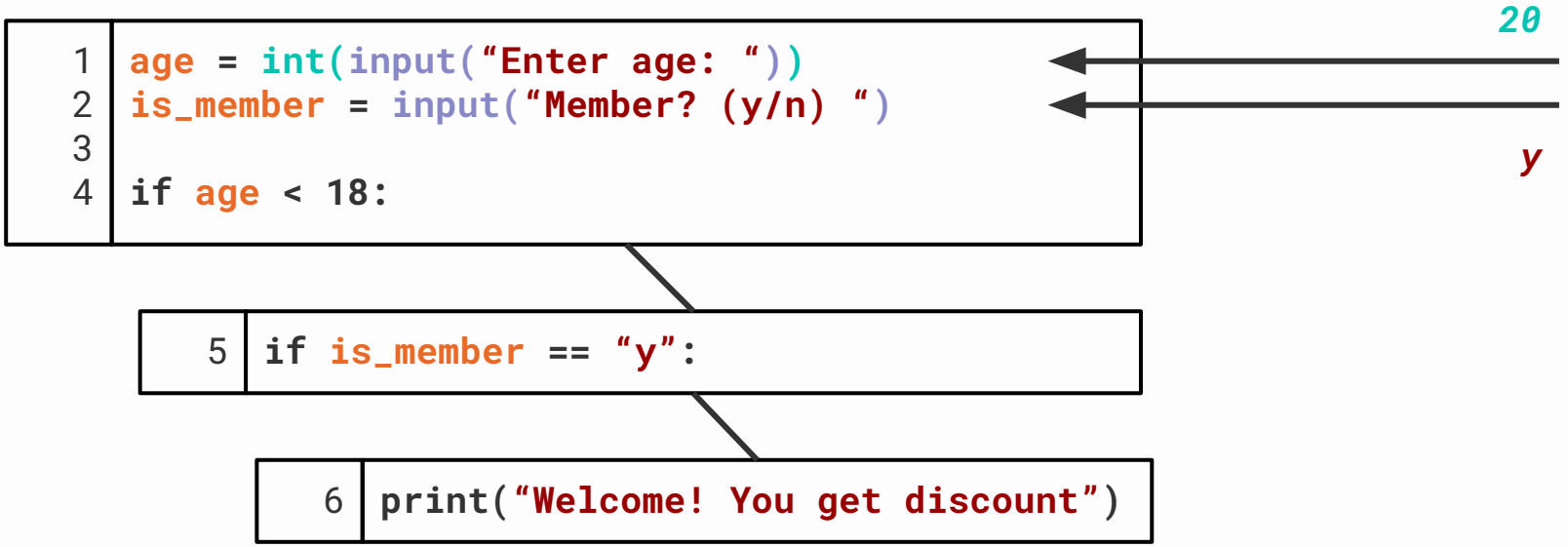
There is no limit on what code can be added inside a conditional - even another conditional!

```
1  age = int(input("Enter age: "))
2  is_member = input("Member? (y/n) ")
3
4  if age > 18:
5      if is_member == "y":
6          print("Welcome! You get discount")
7      else:
8          print("Welcome!")
9  else:
10     print("Not allowed!")
```

20

Nested If-Conditions

There is no limit on what code can be added inside a conditional - even another conditional!



And Operator

You can use the **and** operator to add extra conditions for a single statement:

```
1  if condition_1 and condition_2 and condition_3:
2      # Code
```

```
1  number = int(input("Provide a number"))
2  if number > 0 and number % 2 == 1:
       print("You gave a positive AND odd number")
```

Make a new file and try this code

Quick Exercise: Application

Define the following Boolean variables

```
has_government_id = True
has_nbi_clearance = True
has_registered = True
```

Then print the following if the user has a government ID AND clearance AND registered

```
Processing finished
```

Then, try changing the variables to False and see what happens!

TIP: Make a new file for this

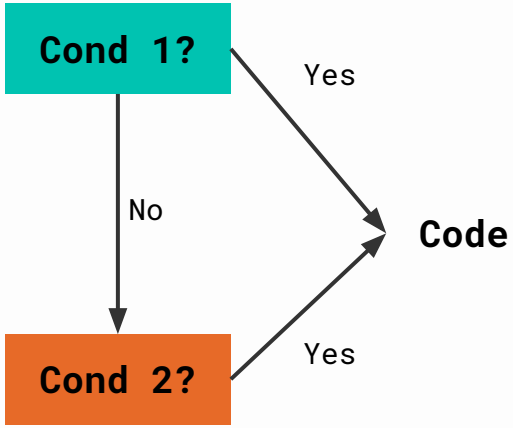
Or Operator

You can use the **or** operator to add alternative conditions:

```
1  if condition_1 or condition_2:
2      # Code
```

```
1  response = input("Continue? ")
2  if response == "yes" or response == "YES":
3      print("We will continue!")
```

Make a new file and try this code



Quick Exercise: Volume Number

Ask the user for a number

```
number = int(input("Provide a number: "))
```

Check if the number is special (**equal to zero, one, five, or ten**),
then print the appropriate result

```
Special Number Detected!
```

```
Not a Special Number!
```

TIP: Make a new file for this

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 divisible_by_three = number % 3 == 0
4 print(not divisible_by_three)
5 print(not number % 3 == 0)
```

Make a new file and try all of these

Quick Exercise: Bad Word

Create a variable with the word the user shouldn't give

```
bad_word = "ice cream"
```

Then ask the user for an input

```
user_input = input("Please provide any input: ")
```

Print the following response based on the input

```
That's a good word!
```

```
That's a bad word!
```

TIP: Make a new file for this

Loops

Don’t repeat yourself anymore

For Range Loop

The for `range()` loop is a way to repeat a process multiple 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
```

Make a new file for the last one and try editing the number!

Quick Exercise: Produce the Following

```
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
This is a very long line that can take some time for you to type.
```

TIP: Make a new file for this

For Range Loop

The true purpose of the range loop is to generate numbers from **0** to the **end-1**

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

```
0
1
2
```

For Range Loop - Manual Equivalent

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

=

```
1  item = 0
2  print(item)
3  item = 1
4  print(item)
5  item = 2
6  print(item)
```

```
0
1
2
```

Quick Exercise: Count to a Hundred

Generate the following numbers per line

```
0
1
...
100
```

Bonus Challenge: Print every even number per line instead:

```
0
2
...
98
100
```

TIP: Make a new file for this

Range() with different start

The `range()` can change where it starts by providing another number first

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

```
1
2
```

Make a new file and try different starts and ends!

Quick Exercise: Skip to a Hundred

Generate the following numbers per line

```
95
96
97
98
99
100
```

TIP: Make a new file for this

Range() with different step

The `range()` can also change how it skips count

```
for item in range(start, end, step):
    print(item)
```

```
for item in range(2, 10, 2):
    print(item)
```

```
2, 4, 6, 8
```

```
for item in range(8, 1, -2):
    print(item)
```

```
8, 6, 4, 2
```

Make a new file and try different starts, steps, and ends!

Quick Exercise: Complete Counting

Generate the following numbers per line

```
5
10
...
100
```

Bonus Challenge: Print every number from one to ten, but in reverse

```
10
9
...
2
1
```

TIP: Make a new file for this

Custom For Loop

In general, for loops are used to iterate or loop through groups of data. You can define your own group by using a square bracket and commas.

```
1  items = ['First Message', 'Second Message', 'Third Message']
2  for item in items:
3      print(item)
```

```
First Message
Second Message
Third Message
```

Make a new file try editing the list!

Custom For Loop - Manual

For loops are mainly used to go through a list of values one at a time.

```
1  items = [
2      'First Message',
3      'Second Message',
4      'Third Message'
5  ]
6  for item in items:
7      print(item)
```

=

```
1  item = 'First Message'
2  print(item)
3  item = 'Second Message'
4  print(item)
5  item = 'Third Message'
6  print(item)
```

Quick Exercises: Bookmarks

Create a list of your favorite sites

```
favorite_sites = [‘facebook.com’, ‘youtube.com’, ...]
```

Then print each of your favorite sites line by line:

```
facebook.com
youtube.com
...
```

TIP: Make a new file for this

Consider the given value

How can we keep asking the user to stop and add the counter?

```
1  counter = 0
2
3  user_input = input("Continue? ")
4  if user_input == 'y':
5      counter = counter + 1
6
7  user_input = input("Continue? ")
8  if user_input == 'y':
9      counter = counter + 1
10 ...
```

Infinite While Loop

While loops can use `True` as a condition to run infinitely.

```
counter = 0


counter = counter + 1
print(counter)
counter = counter + 1
print(counter)
counter = counter + 1
print(counter)
...
```

```
counter = 0

while True:
    counter = counter + 1
    print(counter)
```

When to stop?

Infinite While Loop

While loops can use `True` as a condition to run infinitely.

```
counter = 0
```

```
while True:
```

→

```
counter = counter + 1
print(counter)
```

`counter = 1`

```
while True:
```

→

```
counter = counter + 1
print(counter)
```

`counter = 2`

```
while True:
```

→

```
counter = counter + 1
print(counter)
```

`counter = 3`

Finite While Loop

Make sure to use a condition that will eventually be **False**.

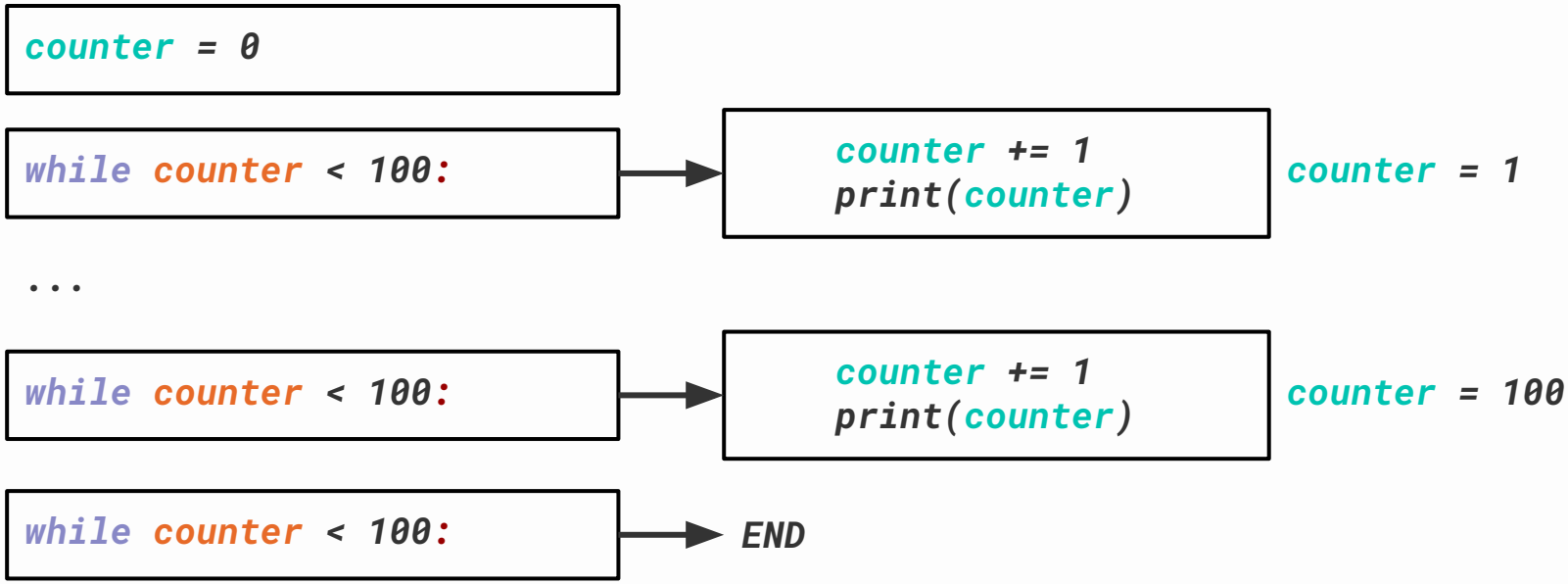
```
counter = 0


counter = counter + 1
print(counter)
counter = counter + 1
print(counter)
counter = counter + 1
print(counter)
...
```

```
counter = 0

while counter < 100:
    counter = counter + 1
    print(counter)
```

Make a new file and try this ^

Finite While Loop

Make sure to use a condition that will eventually be **False**.



While Loop Example 02

While loops are another type of loop that repeats a process while a condition is `True`.

```
counter = 0

user_input = input("Stop?")
if user_input == 'n':
    counter += 1

user_input = input("Stop? ")
if user_input == 'n':
    counter +=  1
...
```

```
counter = 0

user_input = input("Continue? ")
while user_input == 'y':
    counter += 1
    user_input = input("Continue? ")
```

Make a new file and try this ^

Common While Loop Structure

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

```
1  stop_program = False
2  while not stop_program:
3      choice = input("Provide command: ")
4      if choice == "command 1":
5          print("command 1 done")
6      elif choice == "command 2":
7          print("command 2 done")
8      elif choice == "exit":
9          stop_program = True
```

Make a new file and try this

Quick Exercise: Simple Counter

Keep asking for user choice. If user says **add**, increase ***count***.
If user says **minus**, decrease ***count***. If user says **exit**, end code.

```
1  count = 0
2  stop_program = False
3  while not stop_program:
4      choice = input("Provide command: ")
5      if choice == "add":
6          # Add command here
7      elif choice == "minus":
8          # Add command here
9      elif choice == "exit":
10         stop_program = True
```

TIP: Make a new file for this

Loop Breaks

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
```

Make a new file and change condition

```
item = 0
print(item)
if item == 3: False
```

```
item = 1
print(item)
if item == 3: False
```

```
item = 2
print(item)
if item == 3: False
```

```
item = 3
print(item)
if item == 3: True
```

→ break

Quick Exercise: Break on Request

Given the following for range loop, **end the loop immediately if input is y**

```
1  for item in range(100):
2      user_input = input("Stop? (y/n) ")
3      # Add code here to stop if they say y
4      print(item)
```

TIP: Make a new file for this

Continue Keyword

The **continue** keyword skips the succeeding code

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

Make a new file and change condition

```
item = 0
if item == 3: False
print(item)
```

```
item = 1
if item == 3: False
print(item)
```

```
item = 2
if item == 3: False
print(item)
```

```
item = 3
if item == 3: True
print(item)
```

continue →

```
item = 4
...
```

Quick Exercise: Skip Range

Given the following for range loop, **skip printing numbers 20 to 80.**

```
1  for item in range(100):
2      print(item)
```

TIP: Make a new file for this

H2

Cost Calculator v2

Make the previous version more dynamic!

Cost Calculator v2

Ask the user for the number of items to get price:

```
item_count = int(input("Enter number of items: "))
```

Then, depending on the `item_count`, keep asking for that many item price:

```
Input item price:
Input item price:
Input item price:
...
```

Then print the final cost of the items

```
Total Cost: Total cost
```

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 += total
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 += total
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  print(sum(numbers))
```

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

Sum Calculator

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

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

```
5  final_numbers = [1, 2, 3]
6  print(sum(final_numbers))
```

No need to copy paste code

Simple Function Declaration

The default syntax of a function is shown below:

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

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

```
3  greet()
```

Make a new file and try this code

Regular Code Flow

Python code runs line by line from top to bottom

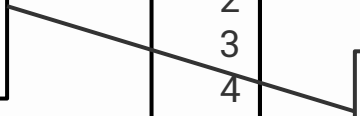
```
1  print(“First Line”)
2  print(“Second Line”)
3  print(“Third Line”)
```

Function Copy-Pasting

When you have a function, it goes back like it's copy pasting the code in-between

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

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



```
1  def extra():
2      print("First Line")
3
4      print("Extra Line 1")
5      print("Extra Line 2")
6
7      print("Second Line")
```

Quick Exercise: Line Generator

Create a function named `line_generator()` that prints the following:

```
Line 0
Line 1
Line 2
```

Then, use the function once:

```
line_generator()
```

TIP: Make a new file for this

Simple Input Declaration

The default syntax of a function with a single input has the following syntax:

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

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

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

Make a new file and try this code

Quick Exercise: Line Generator (Upgrade)

Create a function named `line_generator()` that takes an input number `line_count` and it should print lines depending on the `line_count`

```
Line 0
Line 1
...
```

Then, use the function once with value 4 (you can pick any other number you want)

```
line_generator(4)
```

Multiple Input Declaration

Using more than one input means requires commas

```
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!")
```

Make a new file and try this code

Quick Exercise: Line Generator (Extend)

Create a function named `line_generator()` that takes an input `line_count` and another input `message`. It should print lines depending on the `line_count` and `message`

```
message 0
message 1
...
```

Then, use the function once with value `4` (you can pick any other number you want), and message `"Hello World"` (you can also pick any value you want)

```
line_generator(4, "Hello World")
```

Optional Parameter

You can use a default value for the function inputs

```
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")
```

Make a new file and try this code

Optional Parameter (Overriding)

You can use a default value for the function inputs

```
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(username="Joseph", message="Hajimemashite!")
```

Make a new file and try this code

Quick Exercise: Line Generator (Final)

Create a function named `line_generator()` that takes an input `line_count` and another input `message` . It should print lines depending on the `line_count` and `message.` Make the default value for `line_count` equal to 1 and message to `"Hello World".`

```
message 0
message 1
...
```

Finally, use the function this way (you can override the values if you want)

```
line_generator()
```

Return Value

Functions can return values that can be put in a variable

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

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

Make a new file and try this code

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
```

Why use return functions?

Functions that return values are great to simplify repetitive computations or make it simple

```
1  def distance(x, y):
2      return (x**2 + y**2)**(1/2)
```

```
4  first_distance = distance(3, 10)
5  second_distance = distance(10, 5)
```

Return is Final!

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

```
1  def add(num1, num2):
2      result = num1 + num2
3      return result
4      print(f"The result is: {result}")
```

skipped

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

Make a new file and try this code

Quick Exercise: Number Doubler

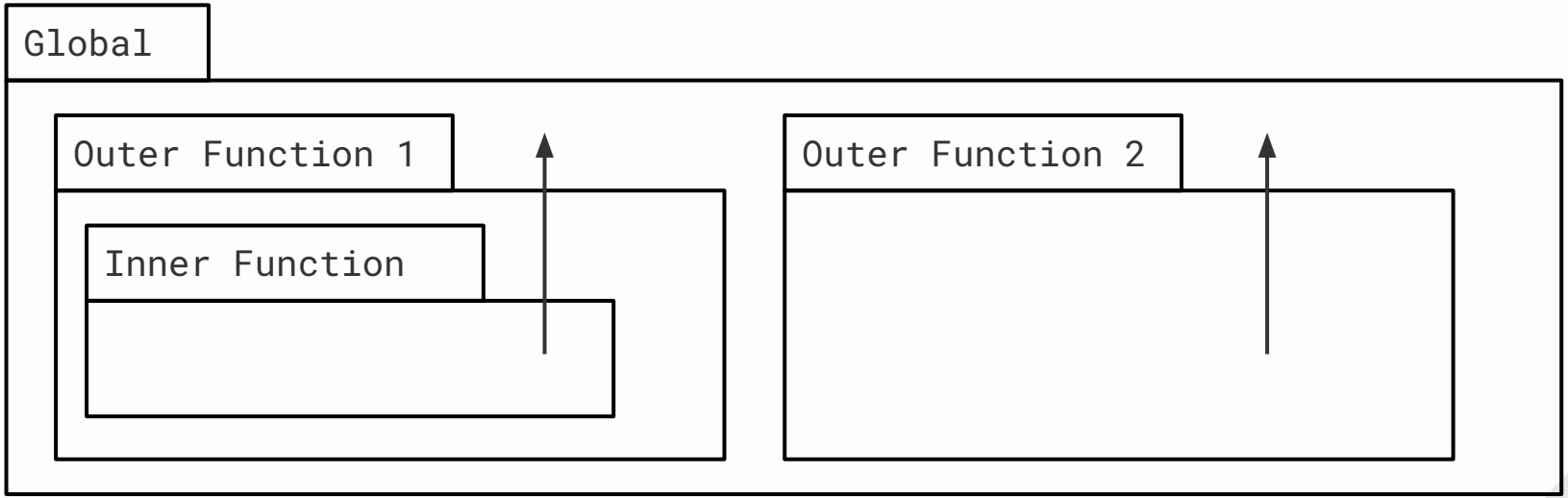
Create a function named ***double()*** that takes an input **number** and return twice the **number**

```
1  def double(number):
2      # Define the code here
3
4  x = 3
5  print(double(x))
```

TIP: Make a new file for this

Function Scoping

The general rule for variable scope is that the variable name is searched starting from the innermost to the outermost



Functions can read outside

Function can detect and print variables outside of it

```
x = 10
def function():
    print("Inner", x)

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

Make a new file and try this code

But functions can’t write outside

Functions can’t change variables outside because this is making another variable with the same name as the one outside

```
x = 10
def function():
    x = 5
    print(“Inner”, x)

print(“Outer”, x)
function()
print(“Outer”, x)
```

Make a new file and try this code

But functions can’t write outside

Even if the variable is given as an input, this does not change anything

```
x = 10
def function(x):
    print(“Inner”, x)
    x = 5
    print(“Inner”, x)

print(“Outer”, x)
function(3)
print(“Outer”, x)
```

Make a new file and try this code

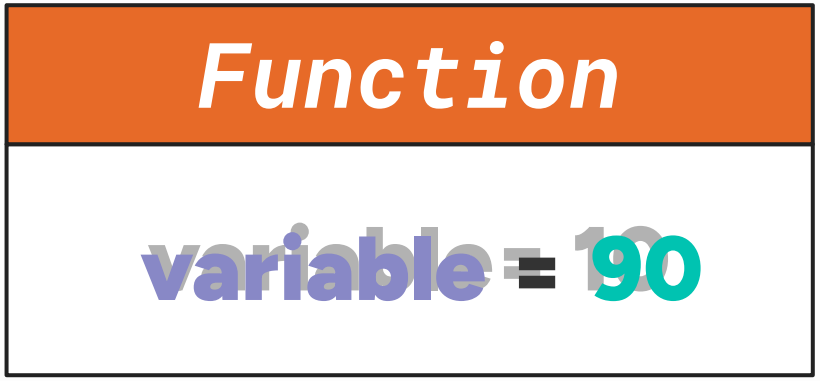
Overwrite using Return

Even if the variable is given as an input, this does not change anything

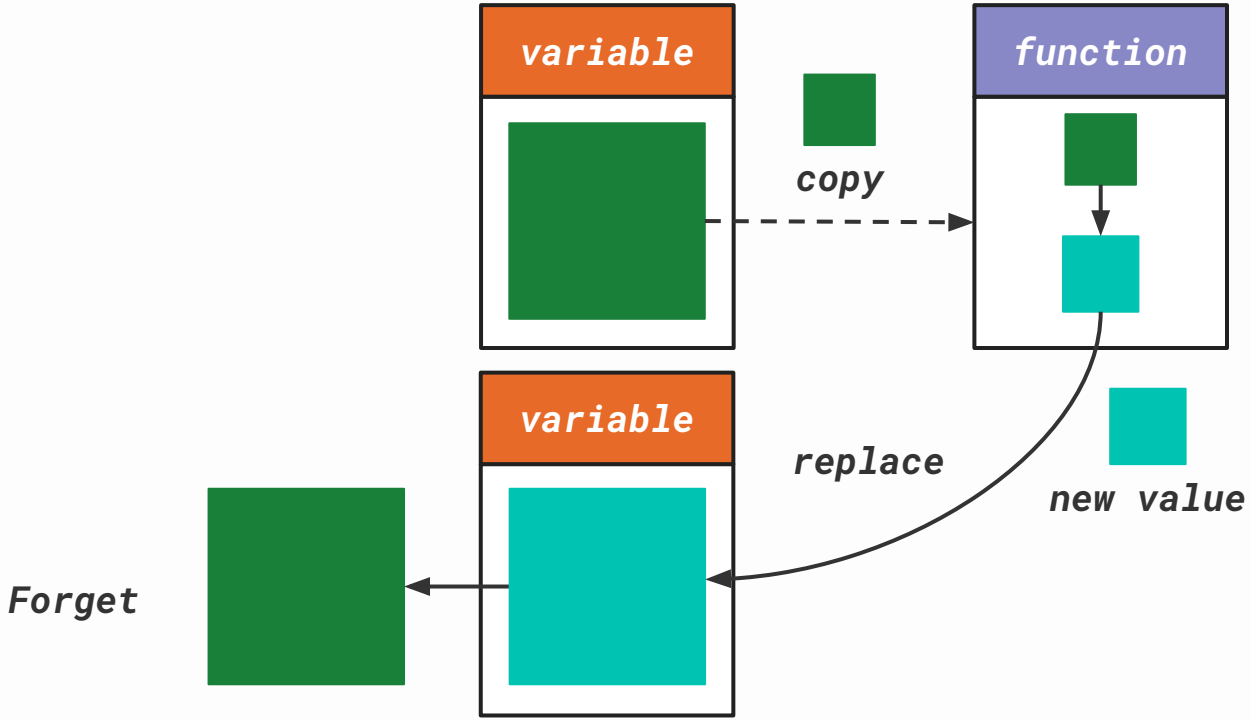
```
x = 10
def function(x):
    x = 5
    return x

print(x)
x = function(x)
print(x)
```

Make a new file and try this code

variable= 10



variable = function (variable)



Conclusions

Avoid Repetition

Use functions to make repetitive processes easier

High-Level First

Other developers don’t need to know all of the details

Convenience

Double check if something similar already exists

Separation of Concern

Group related processes together to organize code

H3

Cost Calculator v3

Clean-up the code for cost-calculator

Cost Calculator v3

```
def get_number_items():
    # Ask user for a valid positive number and return (int)

def get_item_prices(repeat):
    # Ask the user for item prices repeat times and return total (int)

def display(total_price):
    # Print a message about the total_price

def main():
    number_of_items = get_number_items()
    total = get_item_prices(number_of_items)
    display(total)

main()
```

05

Error Handling

Making the code secure by preparing for errors

Possible Errors

Simple mistakes or erratic user input can cause errors in the code

```
print(5 / 0)
```

Exception Catching

To prevent complete stops on an exception, use the `try-catch` statements

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

Make a new file and try either code

Specific Exceptions

You can catch the specific error by adding the name to the right of the except keyword.

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

Error Examples	Description
`TypeError()`	Operation applied to data with the wrong type
`ValueError()`	Function or operation got an inappropriate value
`ZeroDivisionError()`	Specifically occurs when dividing by zero

Quick Exercise: Catch the Error

```
1  number_input = int(input(“Type any number: ”))
2  print(number_input)
```

TIP: Make a new file for this

Multiple Exceptions

The try-except statements can anticipate multiple errors. Checking is by order of listing.

```
# Syntax
try:
    # processes that might cause error
except ExceptionType1:
    # processes on exception 1
except ExceptionType2:
    # processes on exception 2
except ExceptionType3:
    # processes on exception 3
...
```

Multiple Exceptions Example

```
1  try:
2      user_input = int(input("Enter Number: "))
3      result = 5 / user_input
4  except ValueError:
5      print("Input is not a valid number")
6  except ZeroDivisionError:
7      print("Number is a zero!")
8  except KeyboardInterrupt:
9      print("You stopped the code prematurely!")
```

Make a new file and try either code

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  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!")
```

Make a new file and try this code

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      print("Code completed!")
4  except:
5      print("Please don't divide by zero")
6      print("Code completed!")
```

Make a new file and try this code

Full Exception Handling

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

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

Make a new file and try this code

H4

Positive Integer

One of the most common tasks you'll do in the console

Safe Integer

Ask the user for an input that should be a number

```
1 number = input(“Enter number: “)
```

Then try to convert this into an integer using the following:

```
2 number_converted = int(number)
```

But this will cause an error if the user gives a non-integer input. Can you make the error print the following if the input is invalid?

```
Invalid input. Please provide a valid integer
```

Positive Integer

Next, the input should be a POSITIVE integer (greater or equal to zero). If the input is not, print this message instead:

```
Invalid input. Please provide a positive integer
```

Continuous Positive Integer

Finally, keep asking the user for input for as long as they do not give a valid, positive, integer

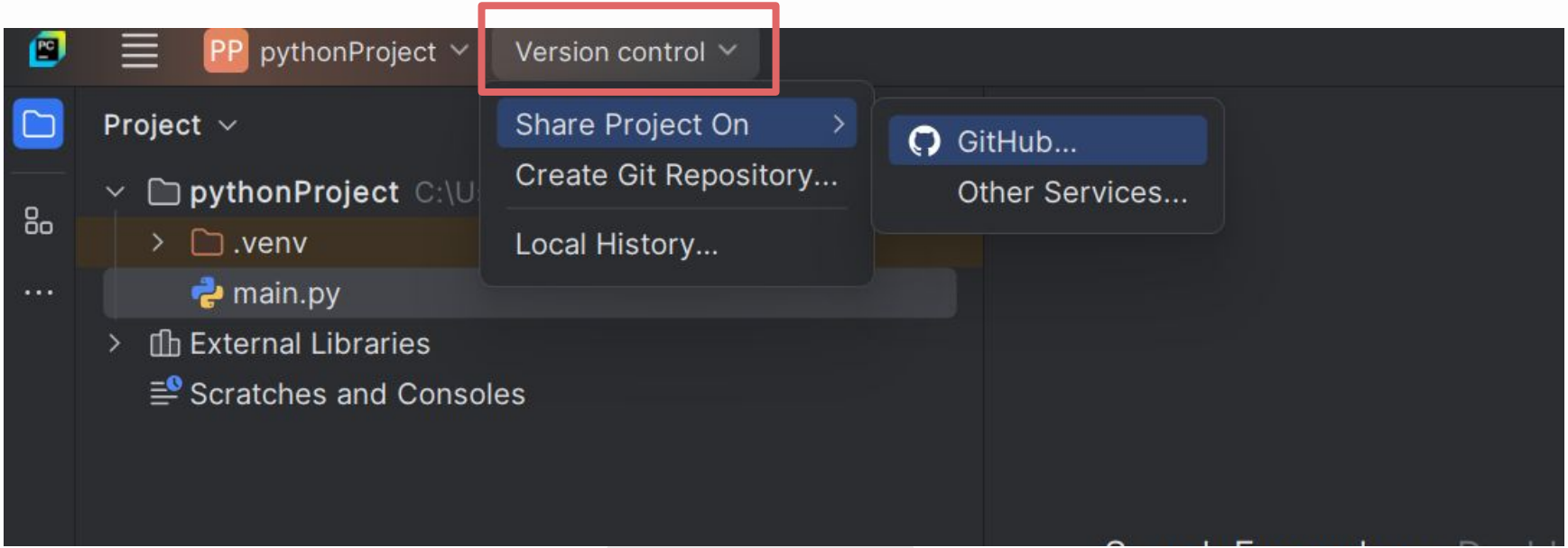
```
Enter number: “Invalid Input 1”
Enter number: “Invalid Input 2”
...
```

Git Setup

Have a reliable way to save and load your files anywhere

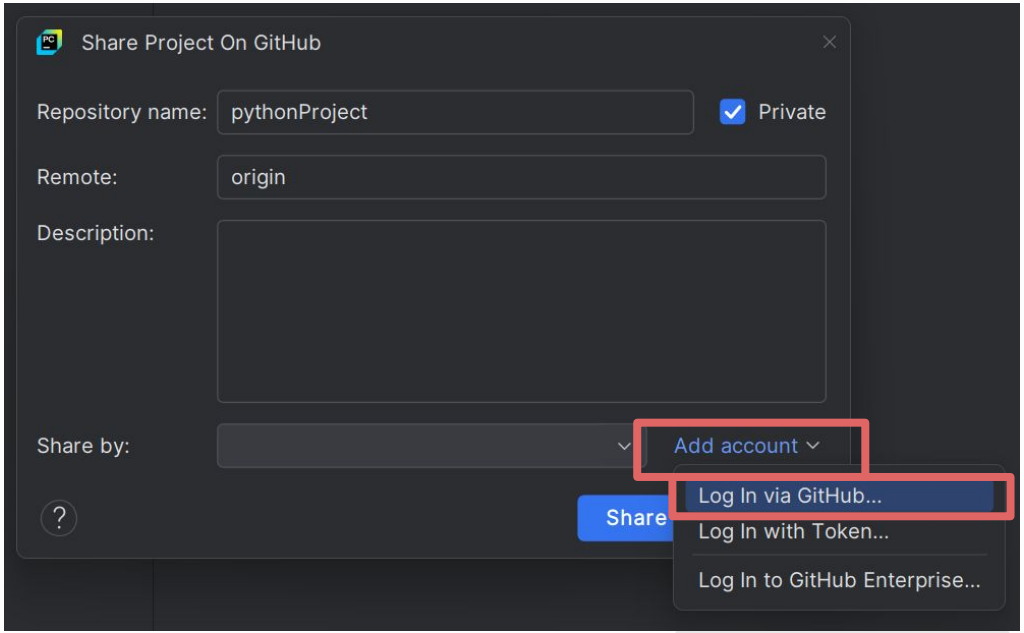
Step 1: Setup your GitHub Repository

At the left side of your project name, select **Version Control**, then select **Share Project On**, and finally select **GitHub**



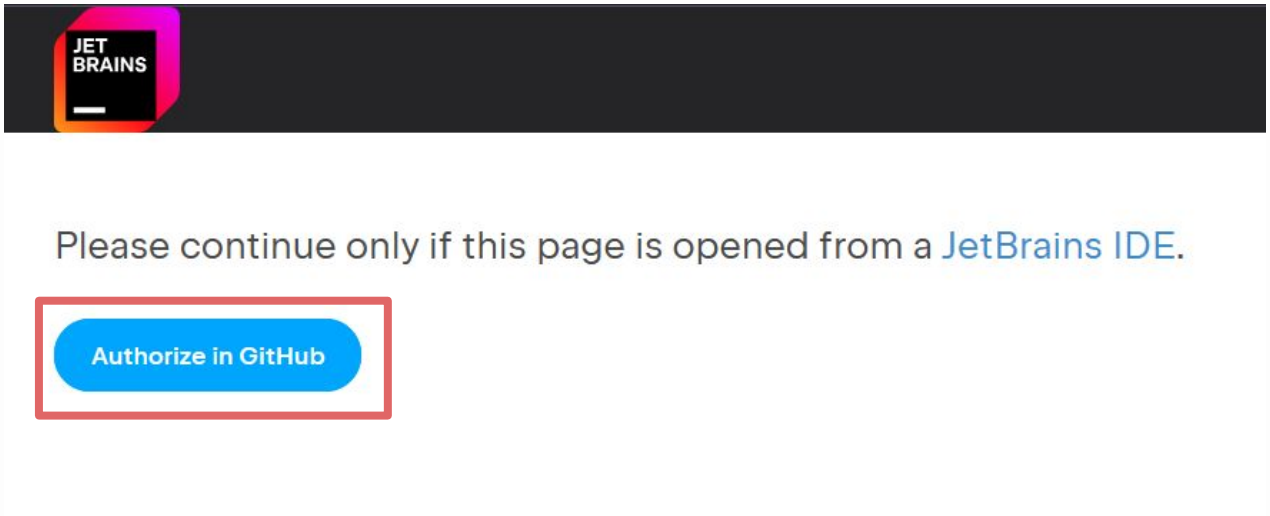
Step 2: GitHub Repository Settings

The previous step will open a pop-up window. Select **Add Account** and **Log in via GitHub**



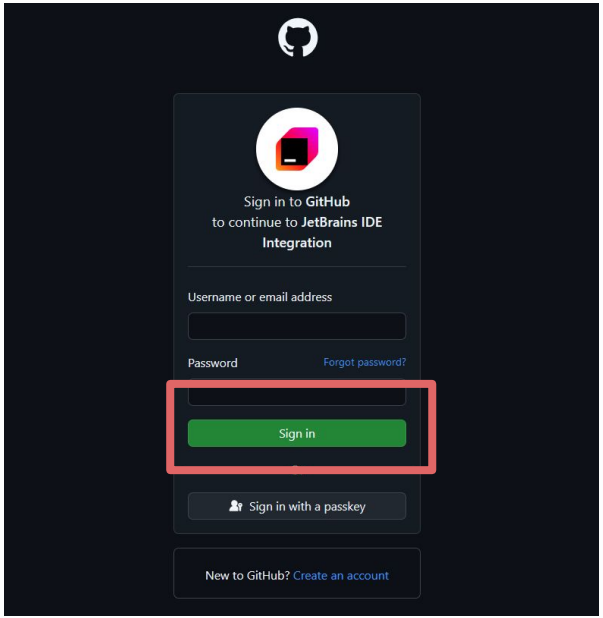
Step 3: Authorize GitHub in JetBrains

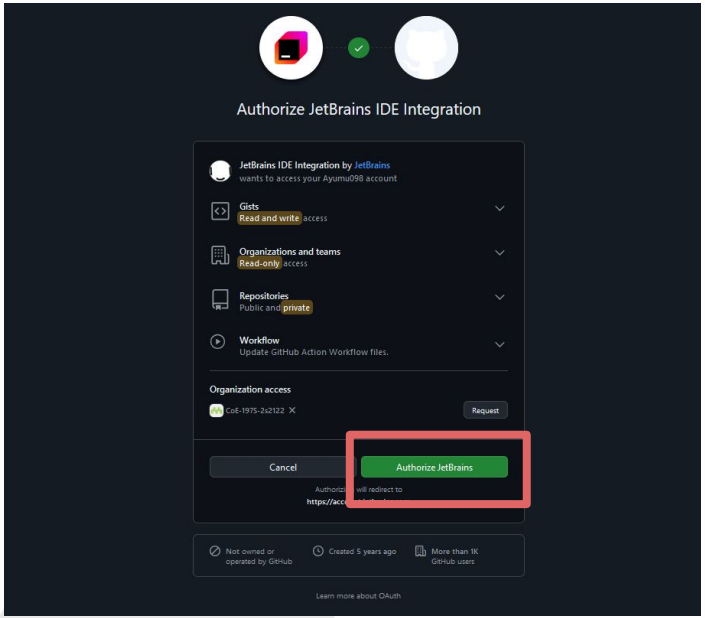
The previous step will open a new window in your browser. Select **Authorize in GitHub**.



Step 4: GitHub Approval

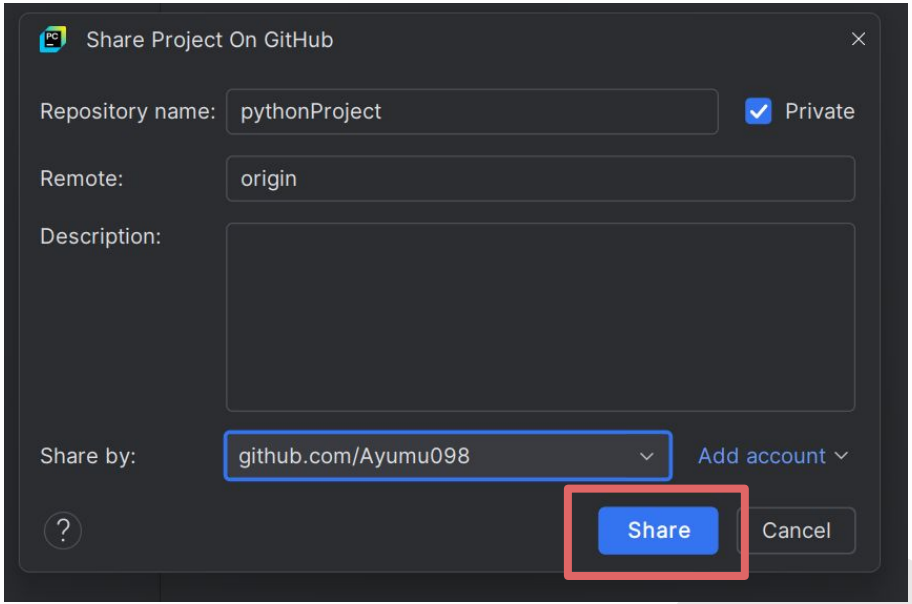
If you have an existing GitHub profile, login. If not, feel free to create a new account.

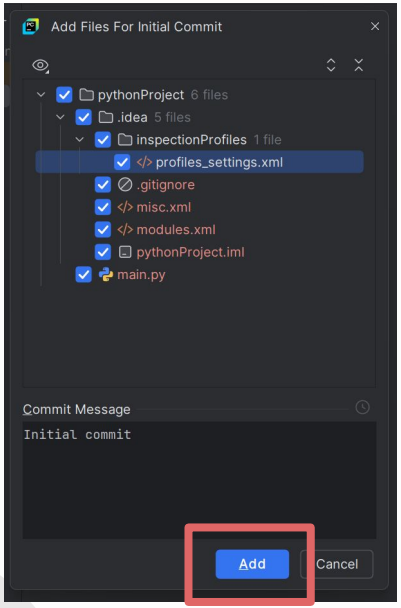




Step 5: Initial Commit

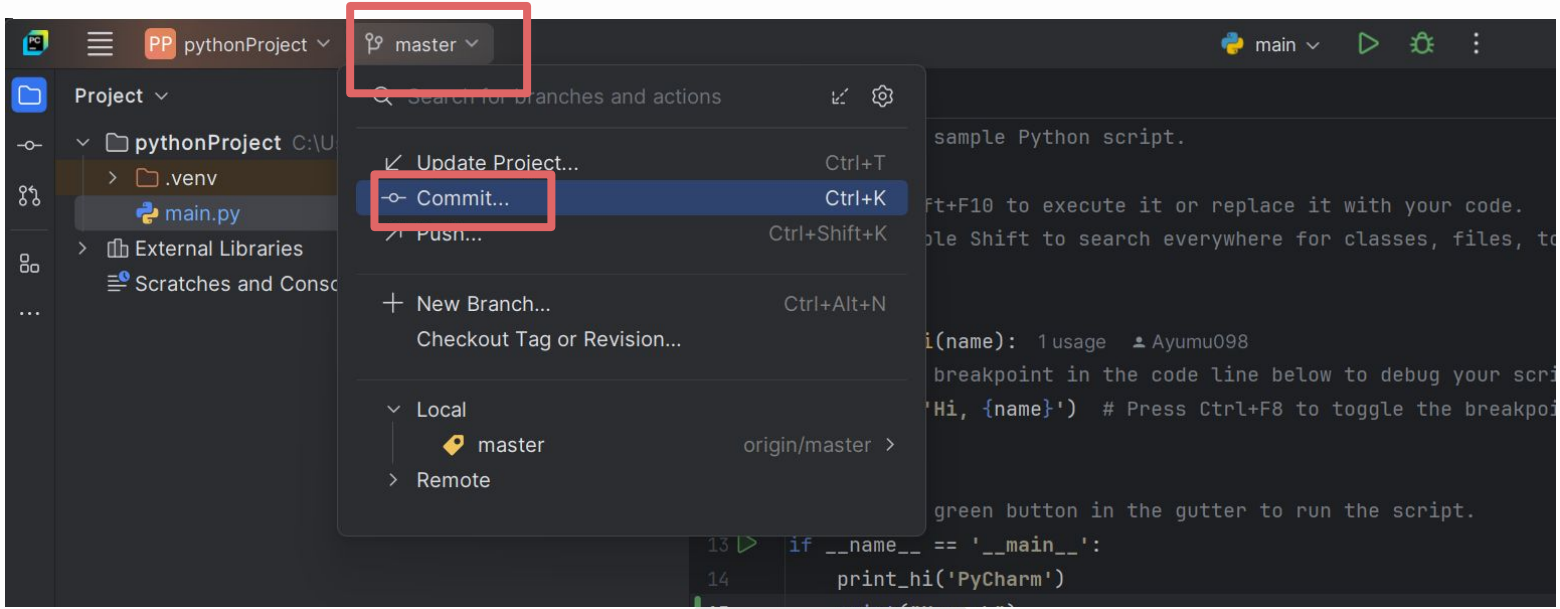
After logging into Github and authenticating, you can now select **Share** in PyCharm and **Add**.





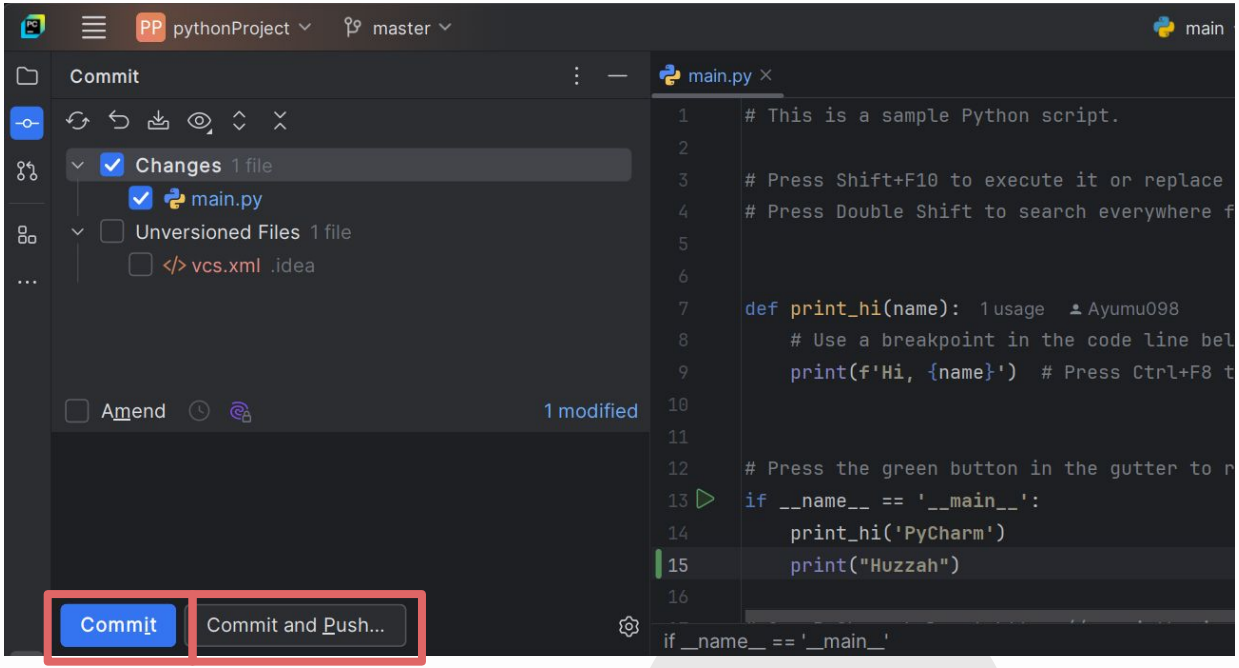
Saving Your Work: Commit and Push

Anytime you make changes to your code, you can select the “master” and press commit.



Saving Your Work: Commit and Push

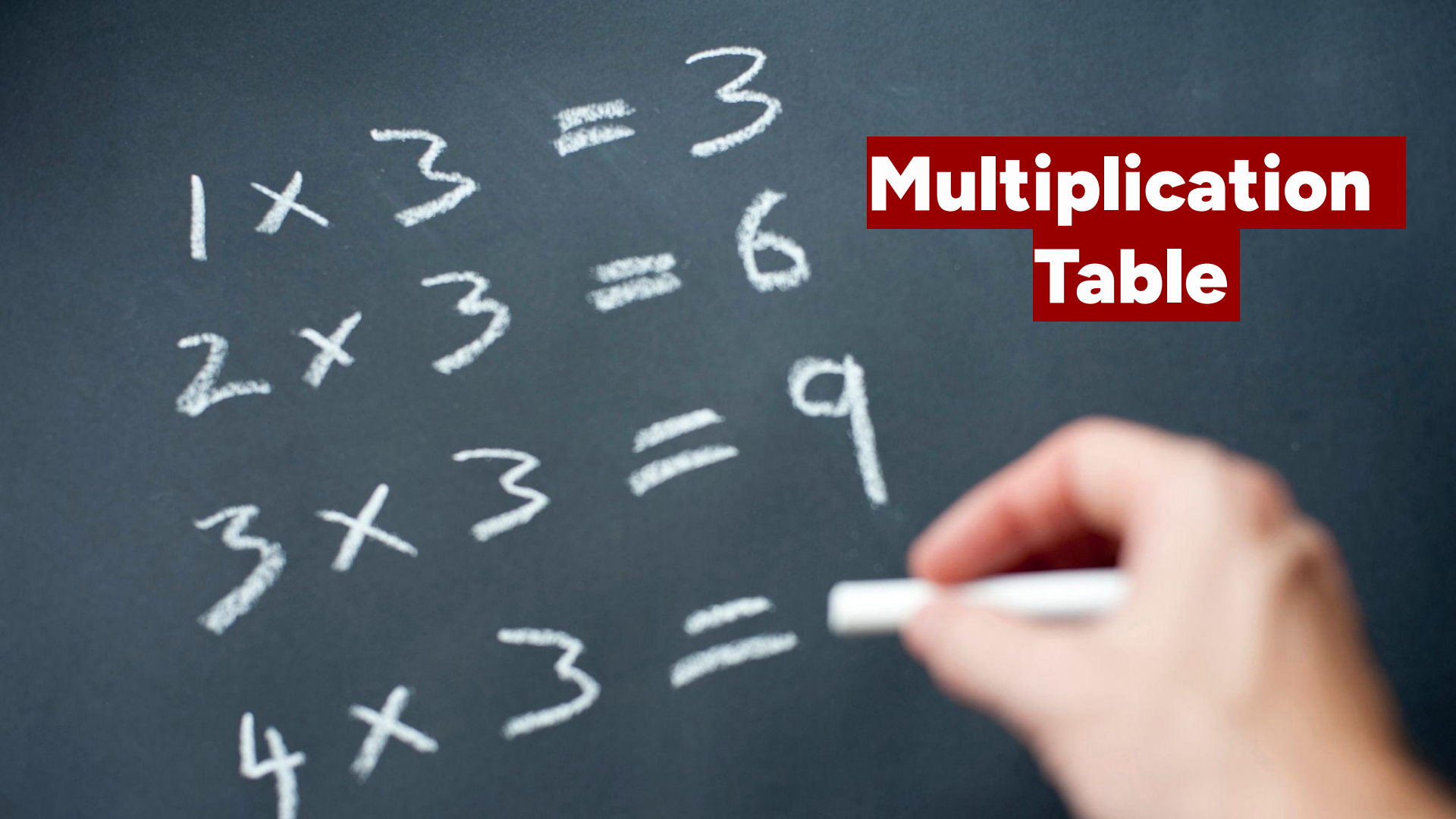
You can select **Commit** to save locally only and **Commit and Push** to save remote as well.



06

Lab Session

Overview of the Course and Python in General


1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 =
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
```

Challenge: Robust Input

Challenge: Alignment

Fizz Buzz

Fizz Buzz: Initial Setup

Ask the user for an integer input

```
1  number = int(input("Pick a number from one to ten: "))
```

Print the integers from 1 to the given `number`

```
1
2
3
4
5
```

Divisible by 3 → Fizz

```
1
2
Fizz
4
Buzz
5
7
8
Fizz
10
11
Fizz
13
14
Fizz
...
```

Divisible by 5 → Buzz

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
...
```

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 )
```

Make a new file and try this code

Recommended Project: Quick Draw

Ask the user for an input

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

Make a random choice for the computer

```
cpu_choice = ...
```

Depending on their choices, tell if the user won, lost, or there was a draw:

```
You win!
```

```
You Lost!
```

```
Draw!
```

Challenge: Robust Input

Challenge: Multi-rounds

Challenge: Two Users

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 and Control Flow