Python: Day 01

Introduction and Basic Syntax

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

Understanding Python

02

Input-Output

Basic data processing

03

Control Flow

Processing Information

04

Functions

Basic Code Organization

05

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 useful features

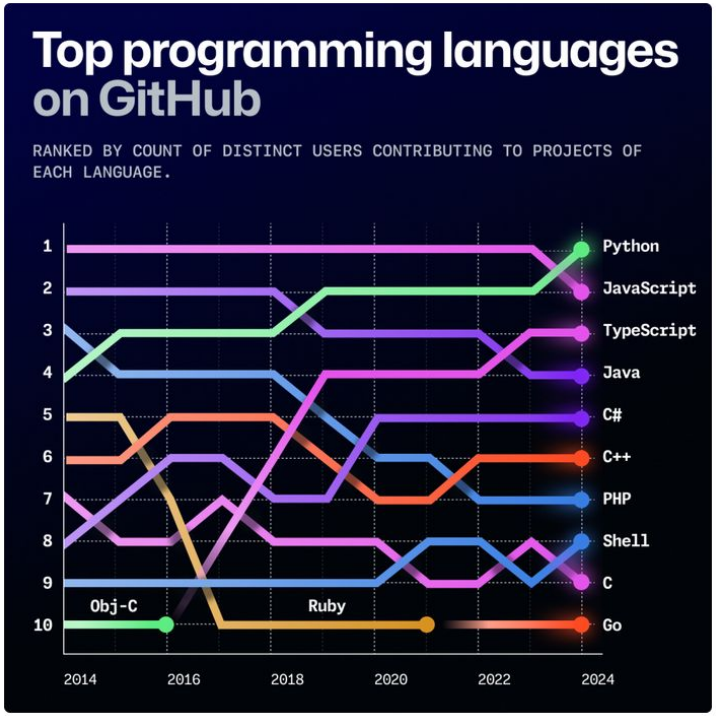
Convenient

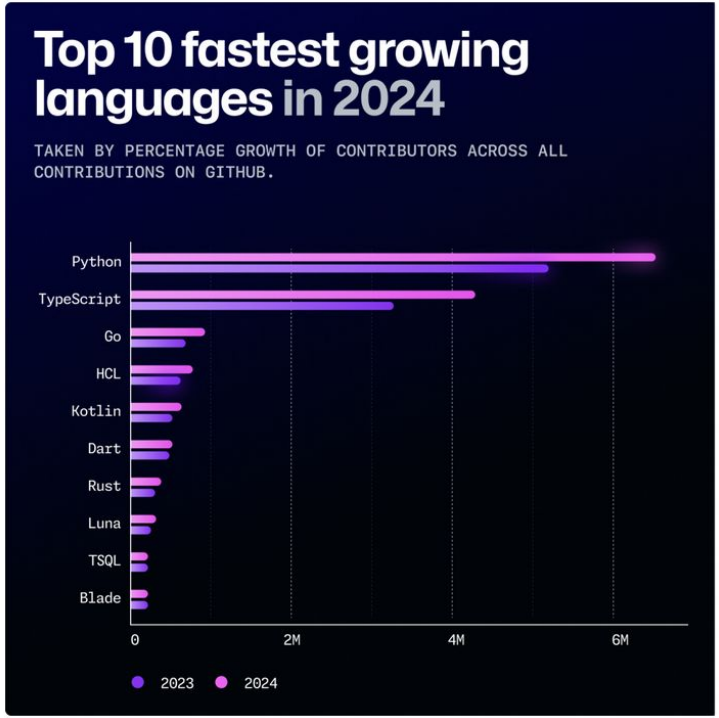
Simple yet diverse for easier development

Active

Large community with a rich ecosystem

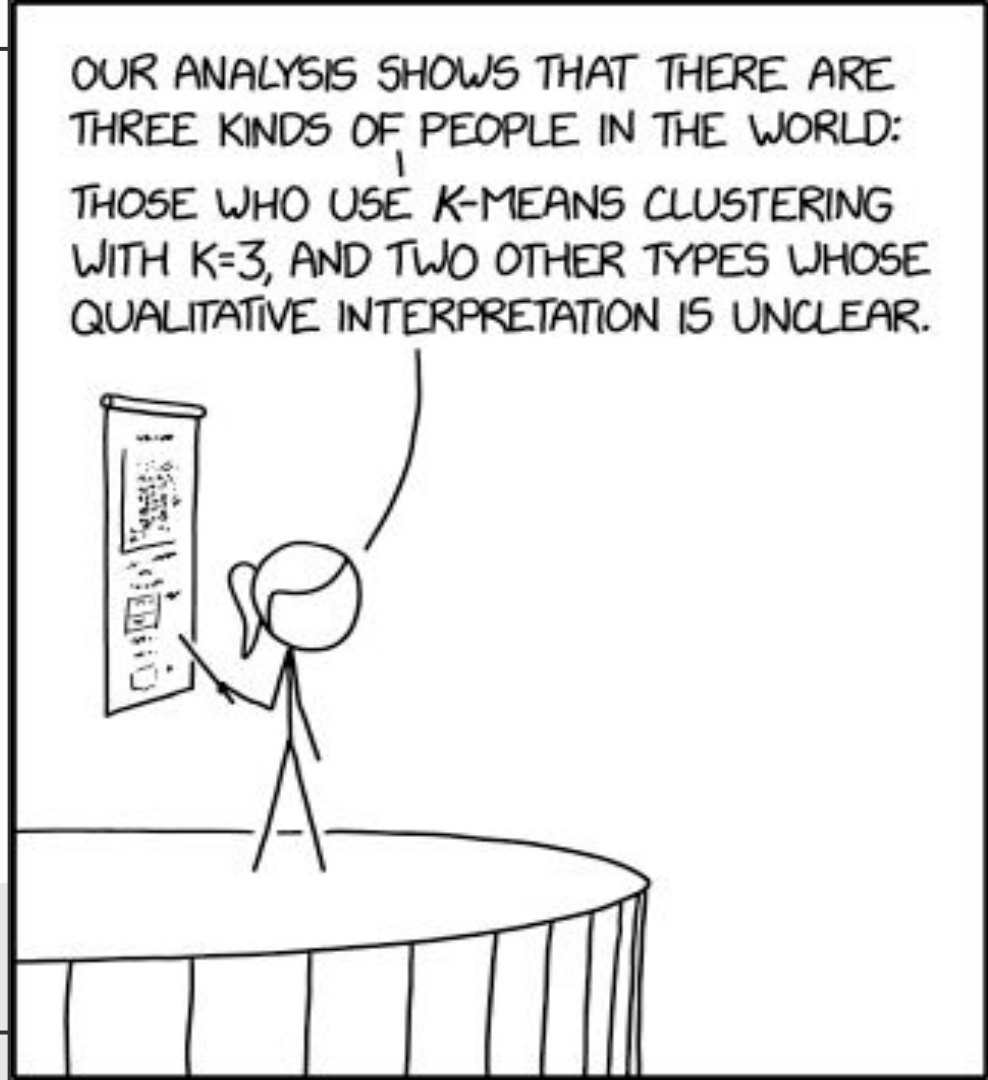
Python Growth





Data Science

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

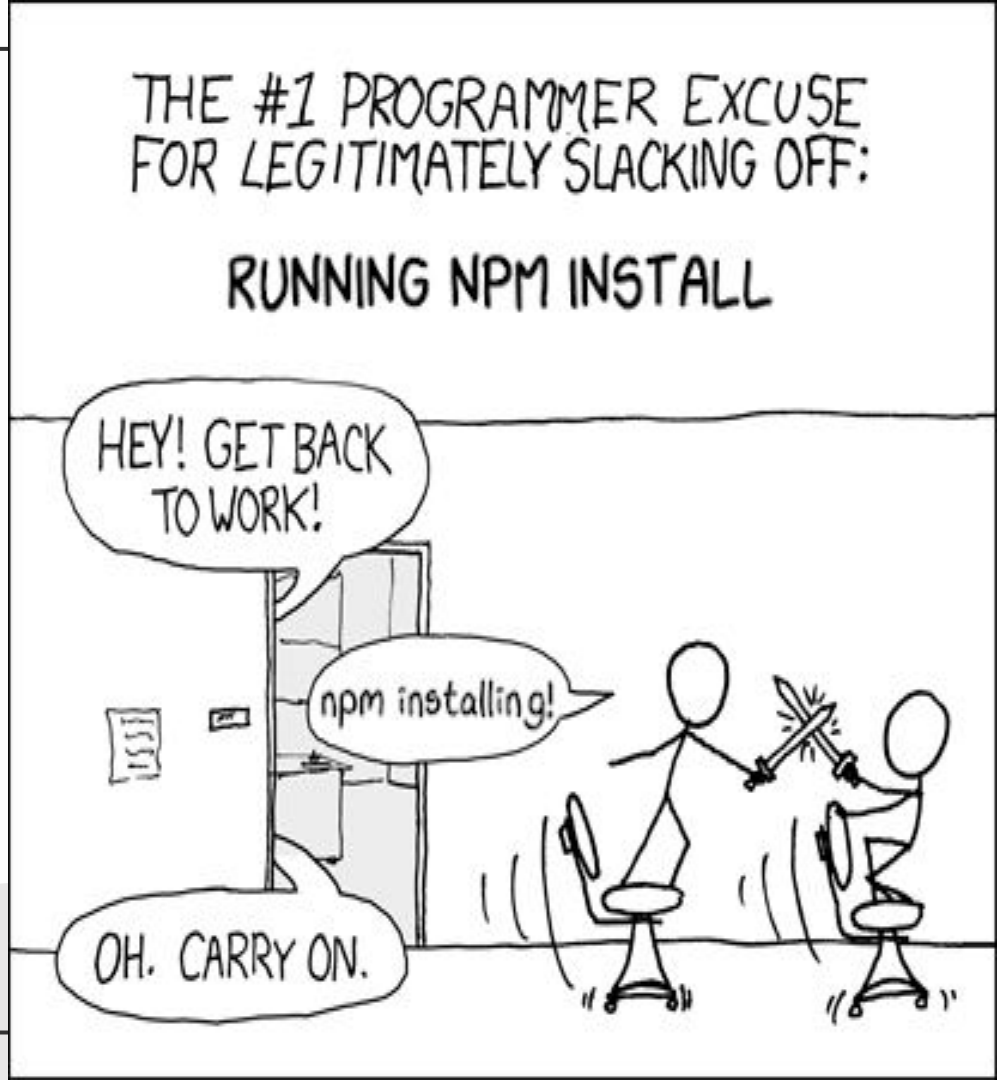


Machine Learning

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

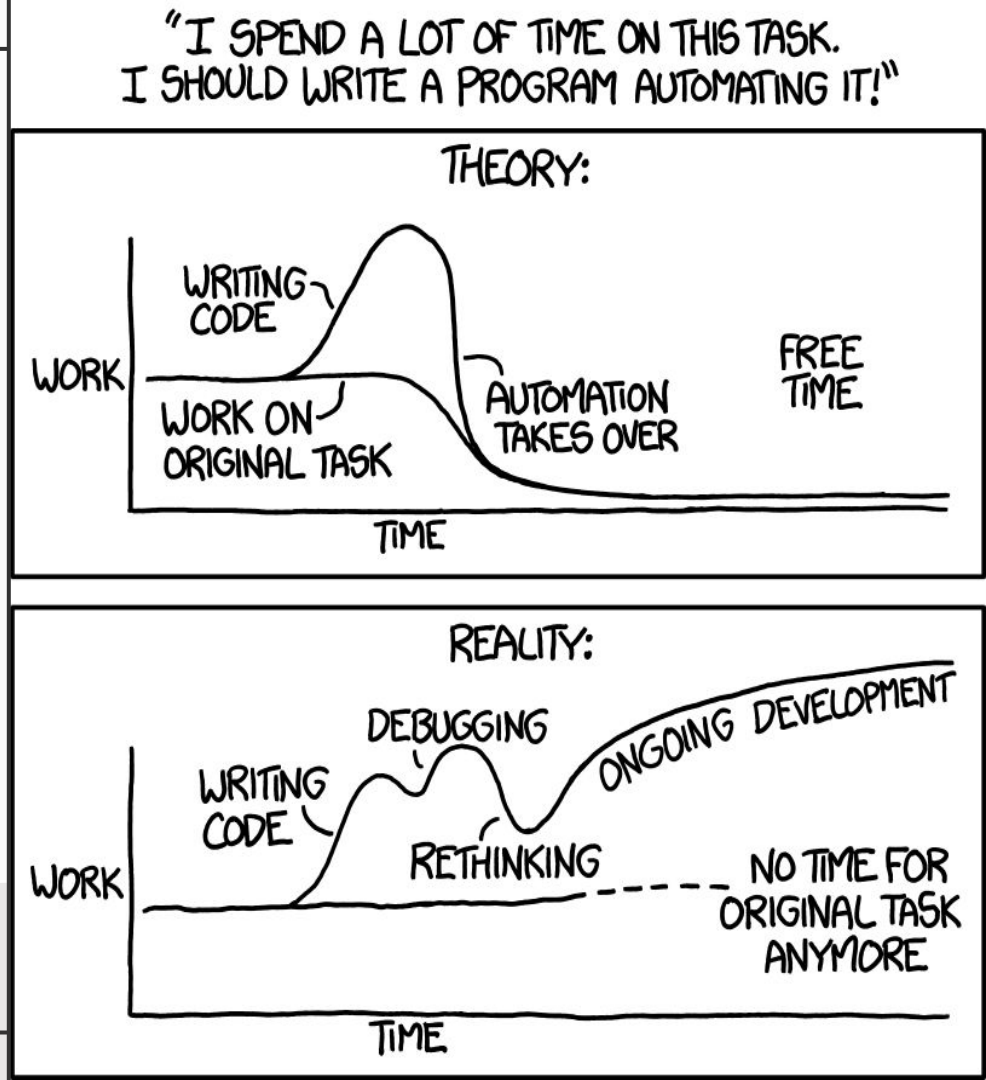
Web Development

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.



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 TV Show: ***Monty Python's Flying Circus***

Fun Fact #2

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

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".

Visual Basic for Applications (VBA)

```
1  Sub HelloWorld()
2      MsgBox "Hello, World!"
3  End Sub
```

Python

```
1  print(“Hello World”)
```

Side by Side Comparison

```
Function r(score As Integer) As String
     If score >= 75 Then
          Result = "Pass"
     Else
          Result = "Fail"
     End If
End Function

Sub CheckScores()
     Dim i As Integer
     Dim score As Integer

     For i = 1 To 5
          score = Cells(i, 2).Value
          Cells(i, 1).Value = r(score)
     Next i
End Sub
```

```
from openpyxl import load_workbook

def r(score):
     if score >= 75:
          Return "Pass"
     else:
          Return "Fail"

wb = load_workbook("scores.xlsx")
s = wb.active

for i in range(1, 6):
    score = s.cell(i, 2).value
    s.cell(i, 1).value = r(score)

wb.save("scores.xlsx")
```

Python Setup

Installing Python from the official source

Python Installation Summary

Here are the key option selections for each prompt during the git installation:

- Components - **Keep as is**
- Default Editor - **Keep as is**
- Initial Branch Name - **Keep as is**
- Path Setup - **Keep as-is**
- SSH Executable - **Keep as-is**
- HTTPS Transport Backend - **Keep as-is**
- Checkout - **Keep as-is**
- Terminal Emulator - **Use Windows Default Console Window**
- Git Pull - **Keep as is**
- Credential Helper - **Keep as is**
- Extra Options - **Keep as is**

Step 1: Download Python Downloader

Go to https://www.python.org/downloads/ and click the first download button. The version and type will be the latest compatible version for your system.

Step 2: Run Python Installer

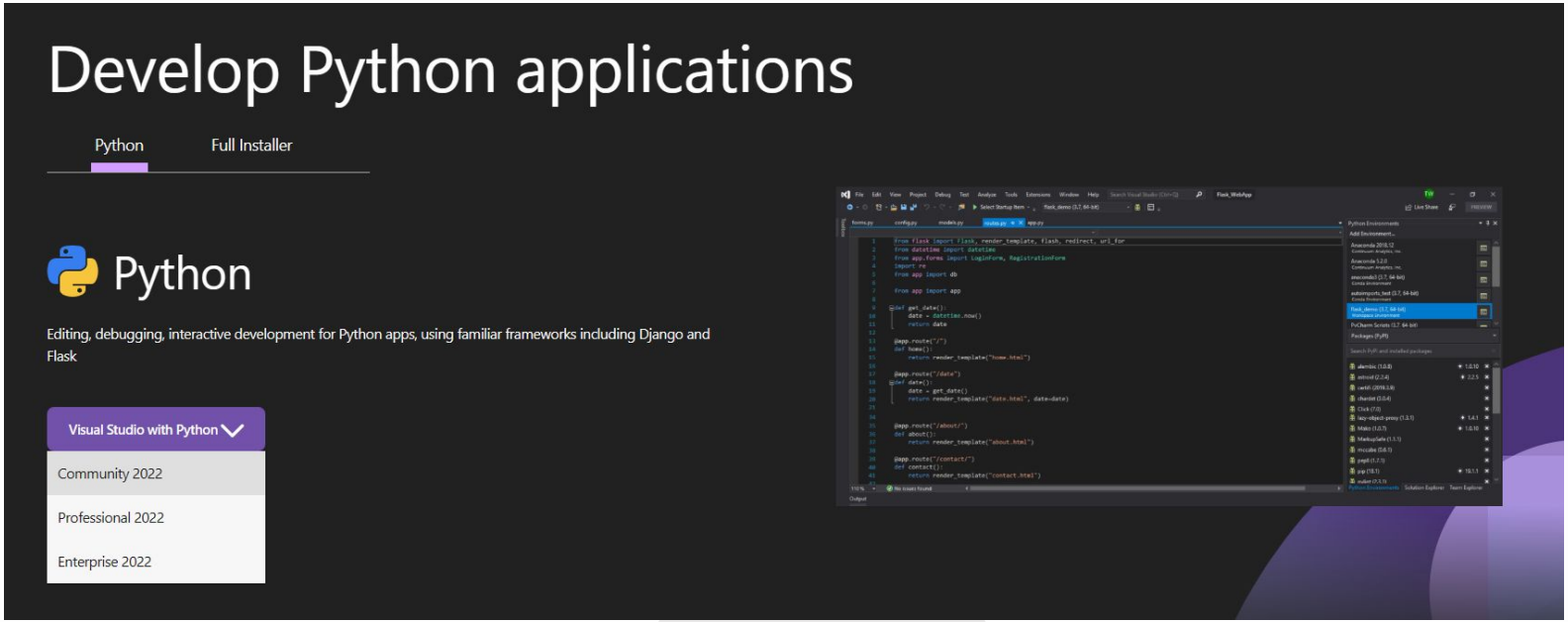
- Run the downloaded installer (preferably with admin privileges)
- Enable all of the checkbox options
- Select Install Now option

Visual Studio

Preparing our Integrated Development Environment (IDE)

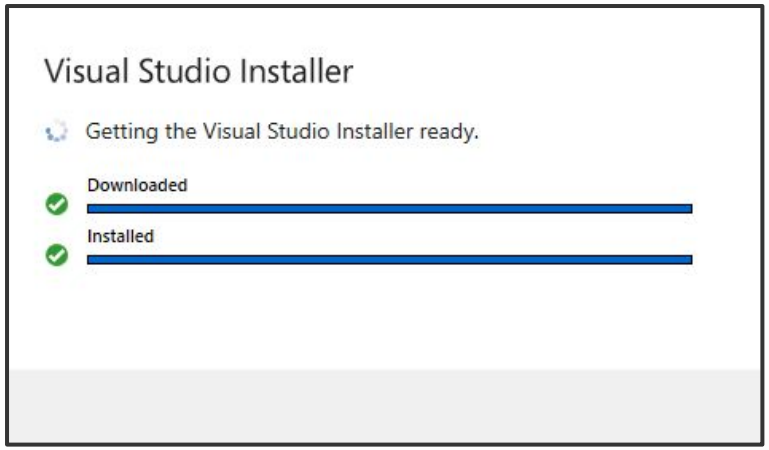
Step 1: Download Python-Based Installer

Go to https://visualstudio.microsoft.com/vs/features/python/ and select the Community Edition



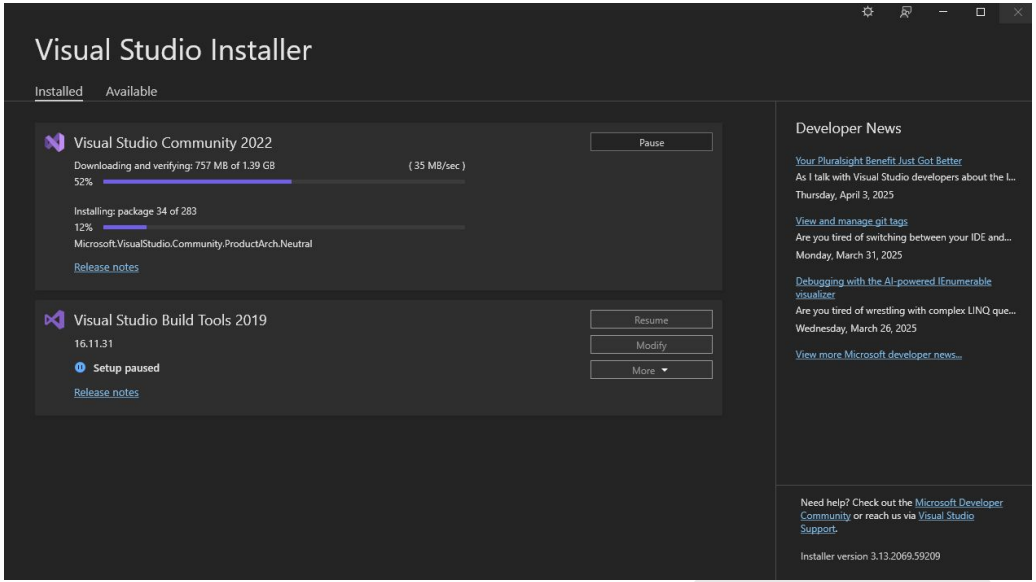
Step 2: Run Python-Based Installer

Upon opening the downloaded file, the following should pop-up first. This will take a few minutes



Step 3: Wait for Visual Studio Code Installer

When the installer automatically closes, it should open this window. This will download for about five to ten minutes, depending on internet speeds

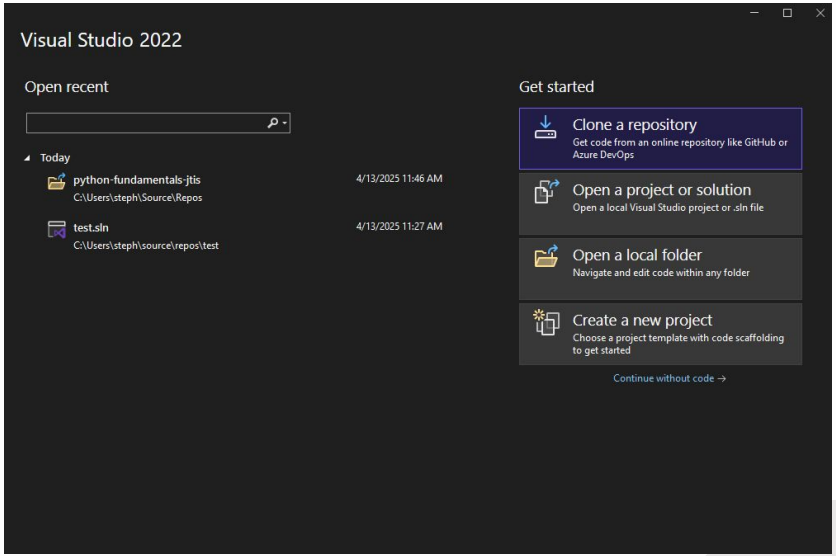


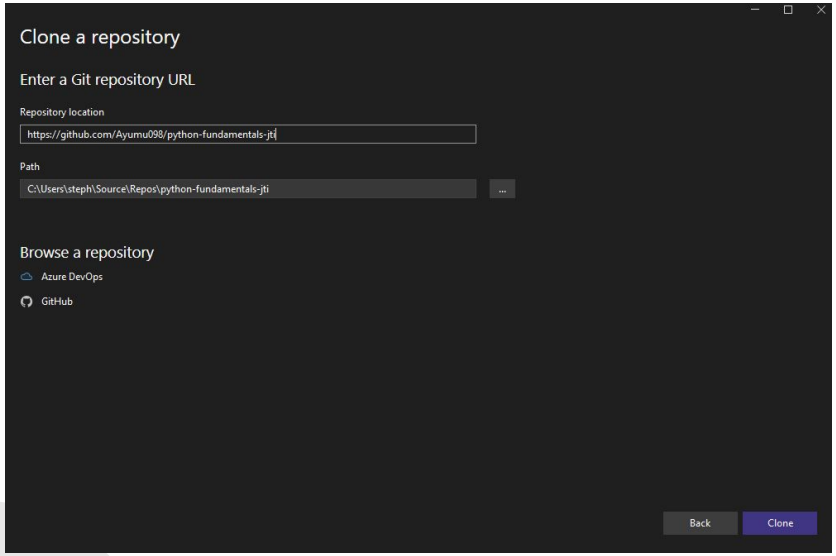
Step 4: Run Microsoft Visual Studio

After the download and installation finishes, manually open Visual Studio 2022 in your computer

Step 5: Clone Repository

Select **Clone a repository** and use this following URL to get a copy of the required materials:
`https://github.com/Ayumu098/python-fundamentals-jti`





Hello World

Understand the basics of printing text

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

Multiple Items, Multiple Lines

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

Multiple Items, One Line

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

Single-Line Comment

To write lines that shouldn't be treated as code, using a pound or hashtag at the start.

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

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

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

If you want to use a comment that spans multiple lines, use a triple quote

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

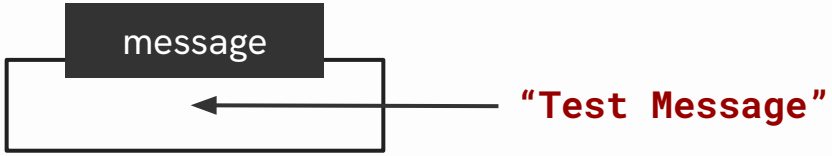
Variables

Representing and storing data in Python

Variable Declaration

Variables are declared using the following format

```
1  message = “Test Message”
```



Variable Printing

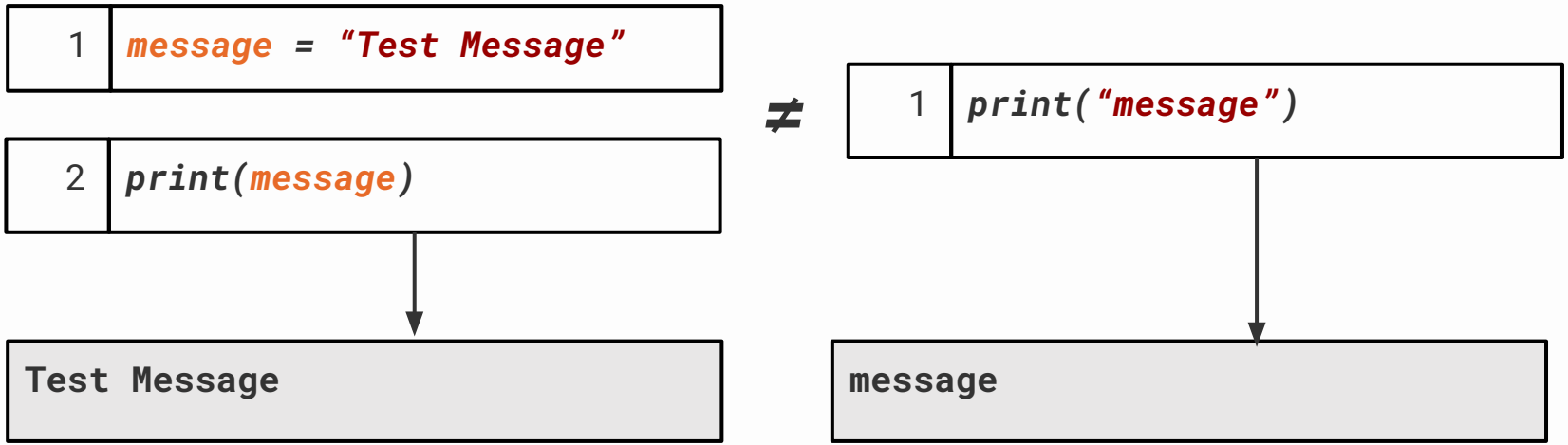
Variables can also be printed with the **`print`** function

```
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


```
print(name)
```

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

Variables and Text

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



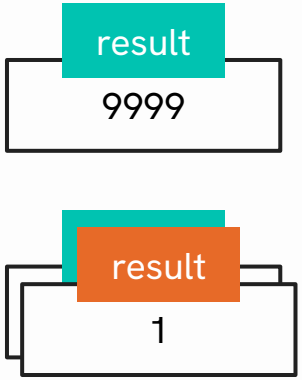
```
message = "Hello! My name is your name."
extra_message = "I am learning Python!"

print(message)
print(extra_message)
```

Variable Reassignment

Variables can be assigned a new value with the same name

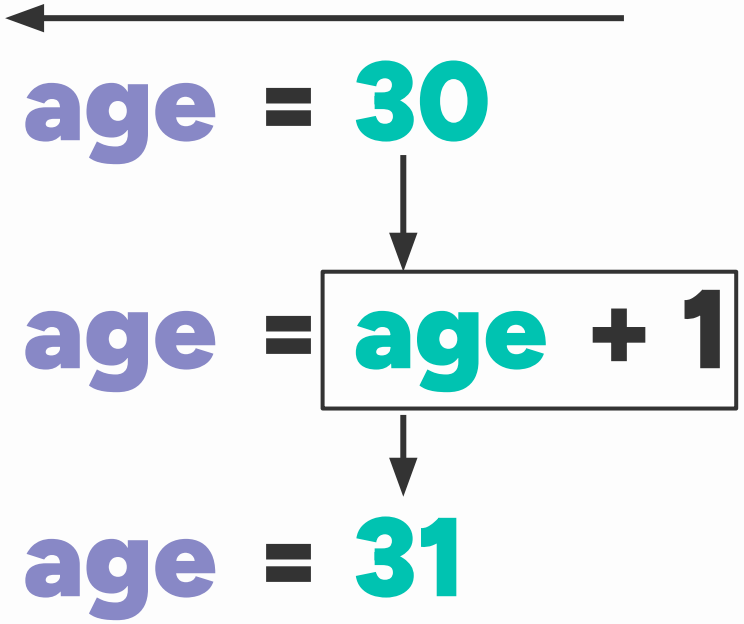
```
result = 9999
print(result)

result = 1
print(result)
```

```
9999
1
```

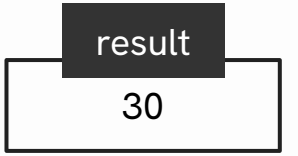


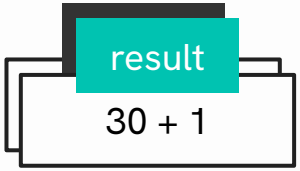
Variable Evaluation

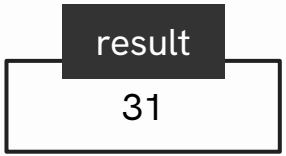
Python evaluates code top to bottom, right to left




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







Quick Exercise: Result Updates

Given the following variable

```
result = 9999
```

Change `result` → `result +1`

```
10000
```

Then, change the `result` → `result *3`

```
30000
```

Variable Name

Learn the rules for naming variables

Variable Naming

- Python supports unicode, allowing most human and special characters
- Variables only support letters or human characters and underscores
- Variables can use digits if they’re not the first character

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  = ””
```

Data Types

The commonly used, built-in types of data

Strings (str)

Strings represent a series of characters or symbols

```
1 empty_string_a = ''
```

```
2 empty_string_b = ""
```

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

Integers (int)

Integers represent whole numbers

```
1 zero = 0
```

```
2 negative_number = -1
```

```
3 large_number = 111_222_333
```

Floating-Point Numbers (float)

Integers represent numbers with decimal places

```
1 zero_float = 0.0
```

```
2 negative_float = -1.2
```

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

Boolean (bool)

Booleans represent answers to Yes-No questions

```
1 is_raining = True
```

```
2 has_red_hair = False
```

None (None)

None represent null or empty values

```
1 example = None
```

Quick Exercise: Task Input

Create the following variables and provide any valid information for a specific task:

```
task_name = What is this task called?
task_cost = How much is required to do this task?
task_difficulty = On a scale of 1-10, how difficult is the task?
task_required =  Is this task required?
```

Then, print each information one line at a time

Print Parameters

Extra features for printing data in the console

Print Separator Parameter

To set the string used when separating multiple items in a `print()`, use `sep` parameter

```
1  print(1, 2, 3, 4)
```

```
1 2 3 4
```

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

```
1234
```

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

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

Quick Exercise: Pointers

Given the following print statement

```
print(1, 2, 3, 4)
```

Use the **sep** parameter to create the following outputs:

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

Print End Parameter

The end parameter sets the final string added after every print (the default is newline "\n")

```
1  print("First")
2  print("Second")
```

```
First
Second
```

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

```
First! Second!
```

Escape Characters

Special characters cause syntax issues so use the escape char `\` to make them valid

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

Quick Exercise: Message Close

Given the following print statement

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

Create the following outputs (without editing the variables):

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

Input Function

Using data not defined in the code

Getting Dynamic Values

input ()

Function — Predefined commands or actions

Parentheses — Marker where `function input` starts and ends

Getting Dynamic Values

```
input("Enter name: ")
```

Function	Predefined commands or actions
Parentheses	Marker where `function input` starts and ends
Text	Text to display while asking for input
Double Quote	Marker where the `text` starts and ends

Getting Dynamic Values

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

Function	Predefined commands or actions
Parentheses	Marker where `function input` starts and ends
Text	Text to display while asking for input
Double Quote	Marker where the `text` starts and ends
Variable	Where the input is stored

Quick Exercise: Favorites

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
```

Type Conversion

Handling data with different types

User Input Issue

What is the problem with this code?

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

Strings and Numbers

Be careful not to confuse integers and strings

```
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

```
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
```

Integer Type Conversion

Convert data to integers 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

Convert data to floats 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

Convert data to booleans with the `bool()` function

Result	Requirement
False	`0`, `0.0`, `""`, `''`, `None`
True	Non-empty string, non-zero number

Boolean Type - Common Pitfalls

Here are common pitfalls when converting to booleans 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

Numerical 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 Operations

Multiple strings can be combined using the addition operator (concatenation)

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

```
Hello World
```

A string can be repeated multiple times using the multiplication operation

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

```
ice ice ice
```

Update Operations

Numerical operations with variable reassignment can be simplified

```
1  result = result + 5
```



```
1  result += 5
```

PEMDAS

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

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

String Formats

Handle strings more conveniently

F-String Formatting

For convenience, variables or code snippets can be inserted directly into strings using the f-string syntax as show below:

```
name ="Juan"
```

```
f"Hello {name}! Nice to meet you"
```

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

F-String Formatting

For convenience, variables or code snippets can be inserted directly into strings using the f-string syntax as show below:

```
f"Calculation: 3 + 5 =  {3 + 5}
```

```
Calculation: 3 + 5 = 8
```

F-String Formatting

For convenience, variables or code snippets can be inserted directly into strings using the f-string syntax as show below:

```
name = input( “Enter name: ” )
nickname = input( “Enter nickname: ” )

print( f“Hello {name} ({nickname})!” )
```

```
Hello Joseph (Jo)!
```

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
```

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!
```

Multiple String Placeholder

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

Quick Exercise: New User

Ask the user for the following input from the user

```
user_name = Input any name here
user_email = Input any email here
user_id = Input any number ID here
```

Then print the following in the console

```
user_name [ user_email ] has been assigned ID user_id
```

Then, ask again for a second user and use the same output template

F1

Email Template

Using placeholders to easily create emails

Follow-Along: Email Template

Dear John Cruz,

I hope you're doing well. I'd love to set up a quick call to discuss the ticket system, which could greatly benefit the customer support team. This feature tracks all outstanding task of the support team, and I believe it could make a significant impact.

Would you be available on April 7, 2025 at 3:00 PM? If not, let me know a time that works for you.

Looking forward to your thoughts.

Best Regards,
{your_name}
{your_position}

Follow-Along: Email Template

```
Dear {recipient_name},

I hope you're doing well. I'd love to set up a quick call to discuss
{feature_name}, which could greatly benefit the {department}. This
feature {use_case}, and I believe it could make a significant impact.

Would you be available on {date} at {time}? If not, let me know a time
that works for you.

Looking forward to your thoughts.

Best Regards,
{your_name}
{your_position}
```

H1

Reimbursement

Practice operations with defined calculations

Hands-On: Reimbursement

Your company allows the reimbursement of four work-related purchases monthly.

```
item1_price = Input first item price here
item2_price = Input second item price here
item3_price = Input third item price here
item4_price = Input fourth item price here
```

The reimbursement system provides 100% discount for the first item, 90% for the second, 70% for the third, and 40% for the fourth item. Calculate the total remaining price.

Challenge: Calculate the savings as well

03

Control Flow

Providing logic to data processing

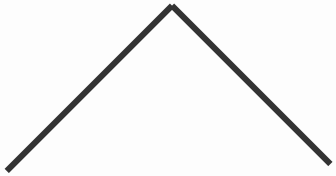

PASS
FAIL

Relations

Checking if two values are related to each other

What are the possible results?

number_1 > number_2

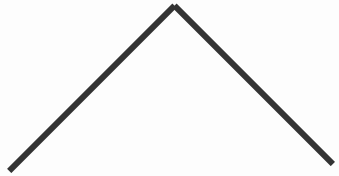


True (Yes)

False (No)

Variable Assignment

result = number_1 > number_2



True
(Yes)

False
(No)

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

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

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
```

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”`

Quick Exercise: Identity Check

Ask the user for the following inputs

```
username1 = Enter first username
username2= Enter second username
```

Then print if the two usernames are the same

```
True or False
```

Conditionals

Control when code executes

If Statement

The if statement allows you to run parts of the code when the given condition is **True**.

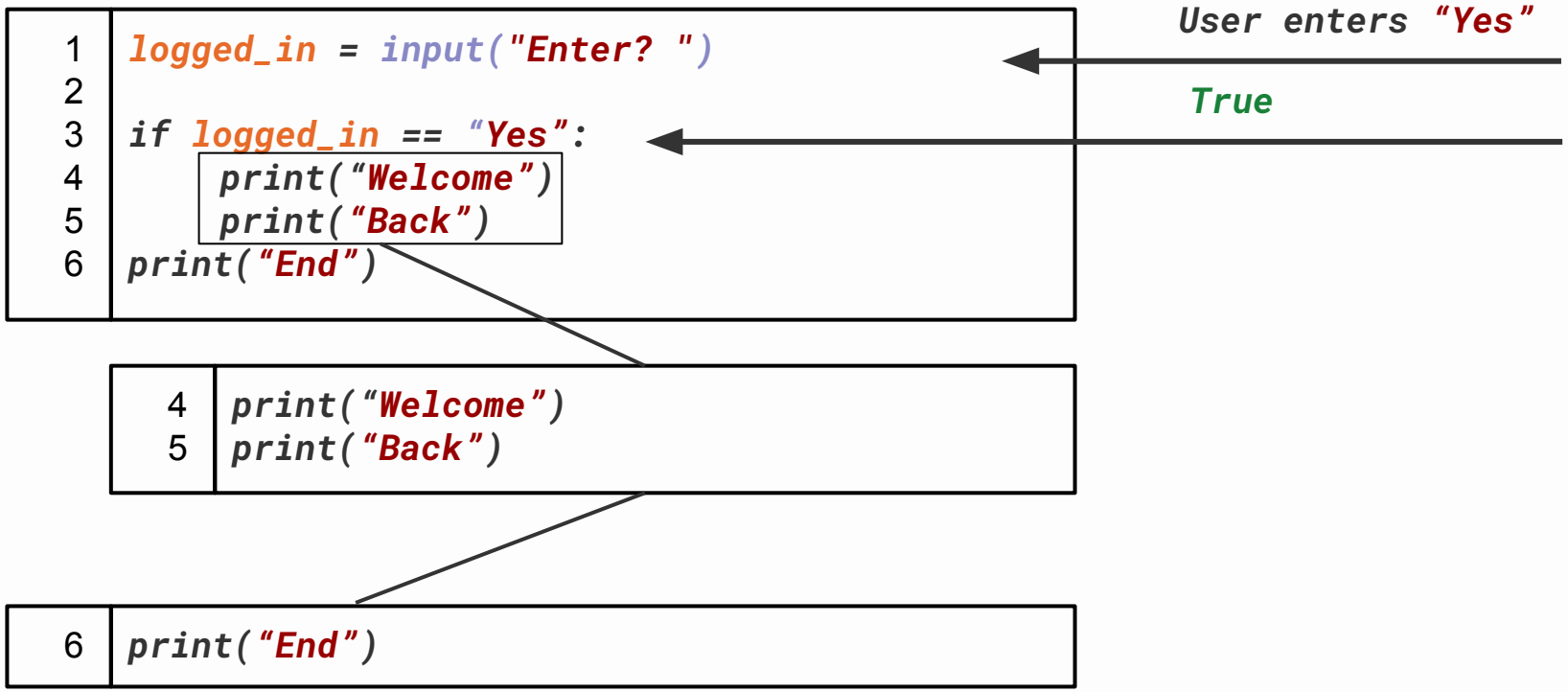
```
1  if condition:
2      """Option code here"""
```

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

If Statement - True

```
1  logged_in = input("Enter? ")
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("Enter? ")
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!
```

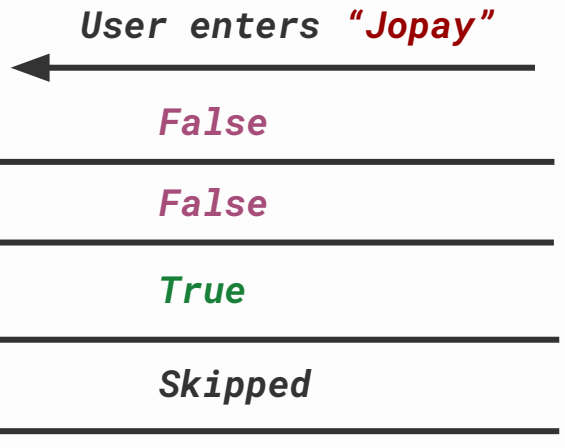
Elif Statement

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

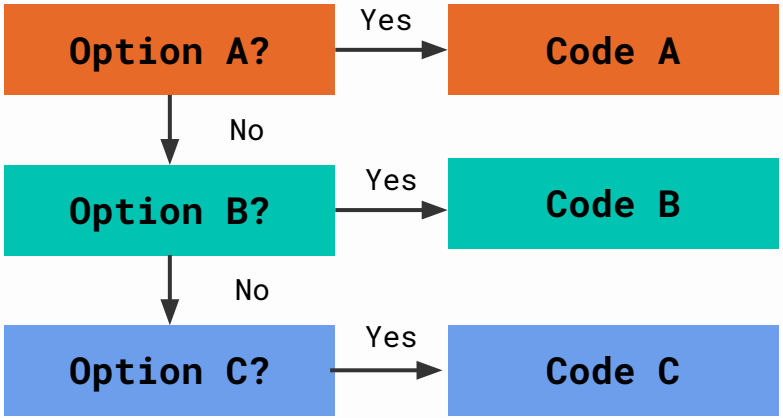
```
1  you_said = input("You said: ")
2
3  if you_said == "Wish":
4      print("107.5")
5  elif you_said == "Hello":
6      print("...it's me")
7  elif you_said == "Jopay":
8      print("...kamusta ka na")
9  elif you_said == "Black Pink":
10     print("...in your area")
```




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

Visualization Conditionals

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



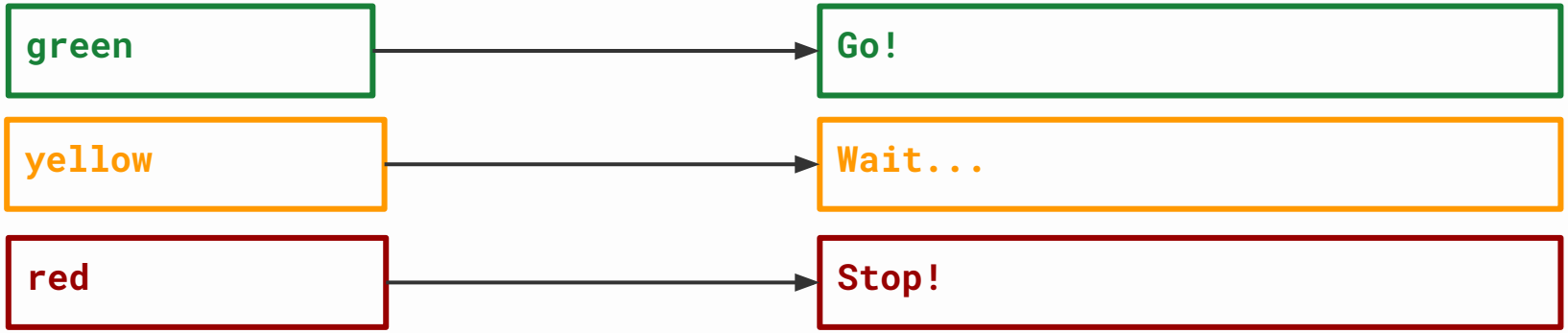

```
if condition_1:
    # Code A

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



Else Statement (Last Resort)

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

```
if condition_1:
    # Option A
elif condition_2:
    # Option B
elif condition_3:
    # Option C
else:
    # 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

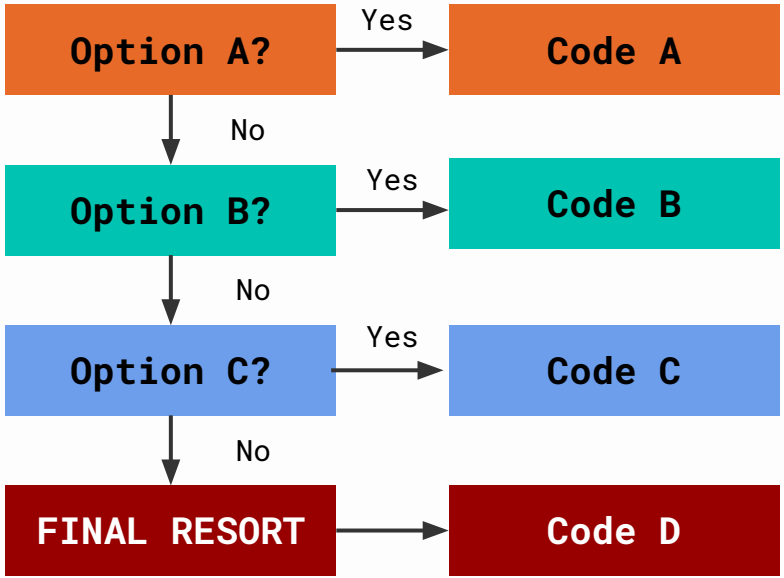
False

FINAL RESORT

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

Visualization Conditionals

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



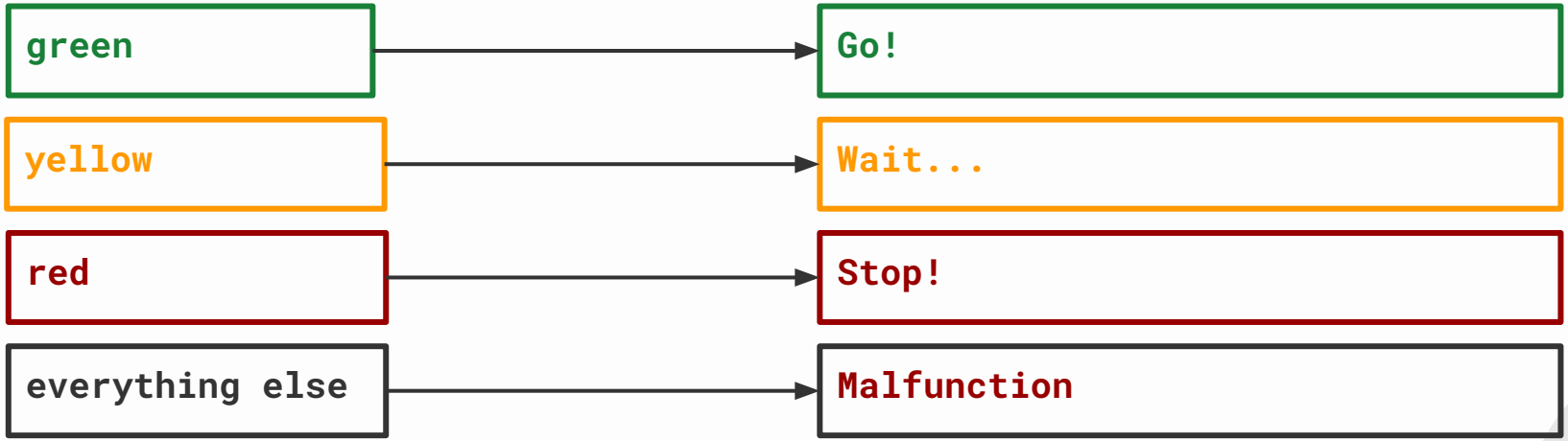

```
1  if condition_1:
2      # Code A
3  elif condition_2:
4      # Code B
5  elif condition_3:
6      # Code C
7  else:
8      # Code D
```

Quick Exercise: Traffic Lights (Complete)

Ask the user input for a color

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

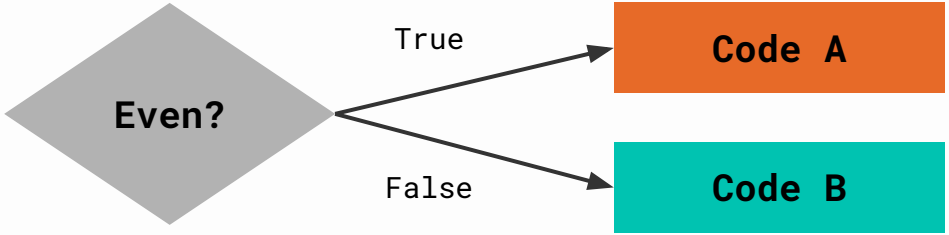
Then print the following depending if the color input



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!
```

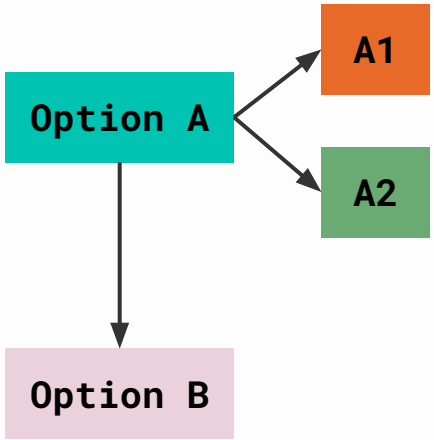
```
Access Denied!
```

+Conditionals

More expressive ways to define logic

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!")
```

Nested If-Conditions

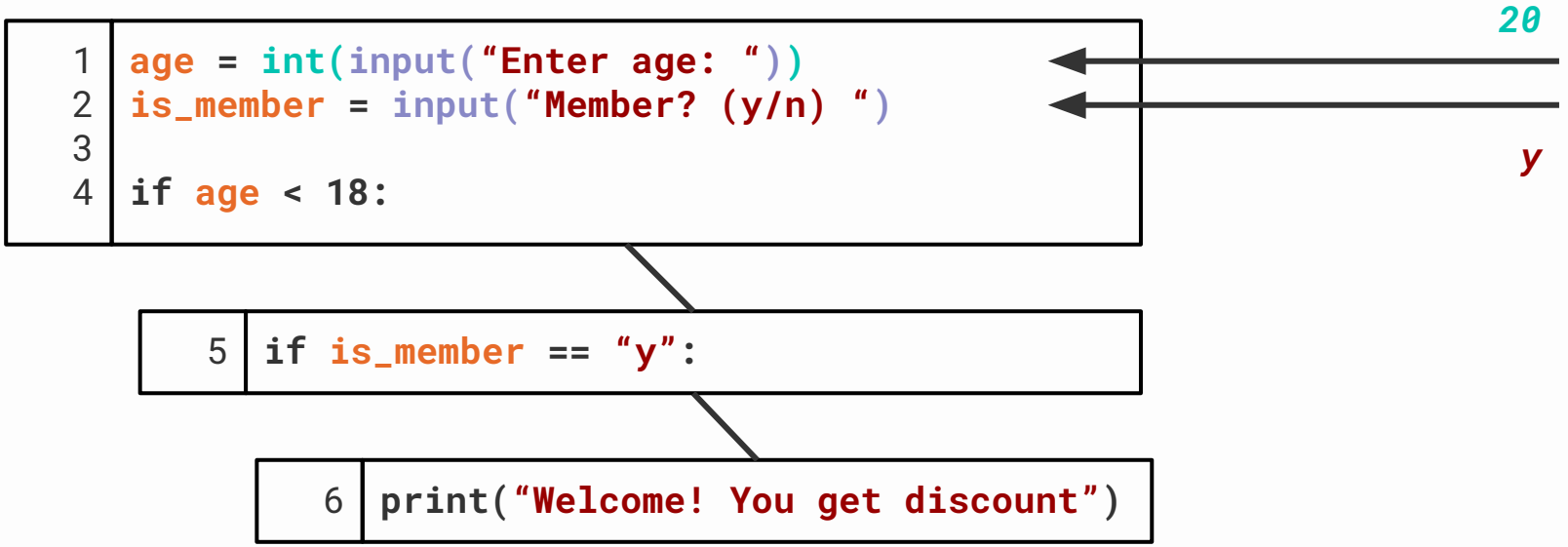
There is no limit on what code can be added inside a conditional - even another conditional!

```
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")
```

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
```

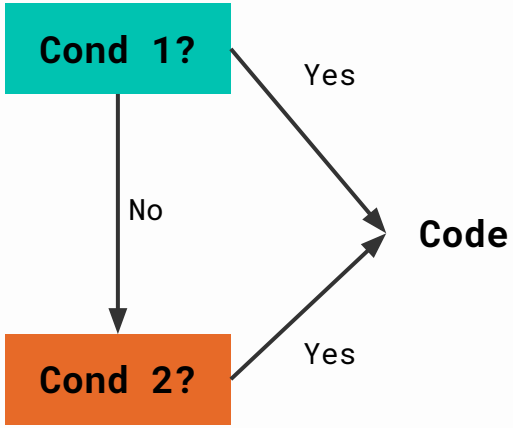
Then, try changing the variables to False and see what happens!

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!")
```



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!
```

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)
```

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!
```

For Loops

Fixed repetition

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
```

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.
```

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
```

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
```

Quick Exercise: Skip to a Hundred

Generate the following numbers per line

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

Range() with different step

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

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

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

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

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

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

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
```

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
```

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
...
```

While Loop

Conditional Looping

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

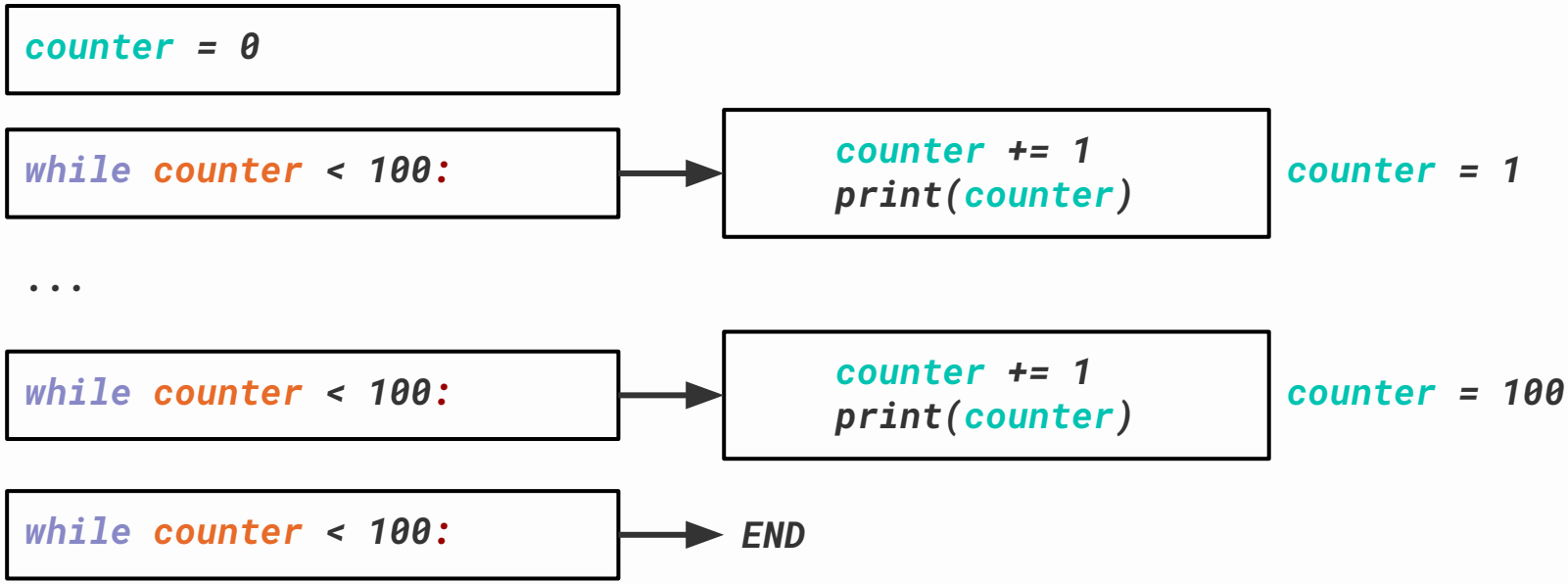
Make sure to use a condition that will eventually be `False`.

```
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)
```

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? ")
```

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
```

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
```

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
```

```
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)
```

Continue Keyword

The **continue** keyword skips the succeeding code

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

```
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)
```

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")
```

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)
```

Multiple Exceptions

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

```
1  try:
2      # processes that might cause error
3  except ExceptionType1:
4      # processes on exception 1
5  except ExceptionType2:
6      # processes on exception 2
7  except ExceptionType3:
8      # 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!")
```

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!")
```

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!")
```

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!")
```

F2

Positive Integer

One of the most common defensive programming task

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”
...
```

H2

Reimbursement v2

Practice more dynamic operations

Hands-On: Reimbursement

Good news! Your company now allows the reimbursement of more than five items.

```
item_count = Input number of items to reimburse
```

Based on the item_count, continually ask for item values

```
Input first item price here
Input second item price here
Input third item price here
...
```

The reimbursement system has changed. It starts at 100%, then 90%, 80%, until it reaches 0%

04

Functions

First step to code organization

Function Definition

Creating custom commands or pre-defined code

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))
```

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()
```

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()
```

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")
```

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!")
```

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")
```

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!")
```

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)
```

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)
```

Quick Exercise: Number Doubler

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

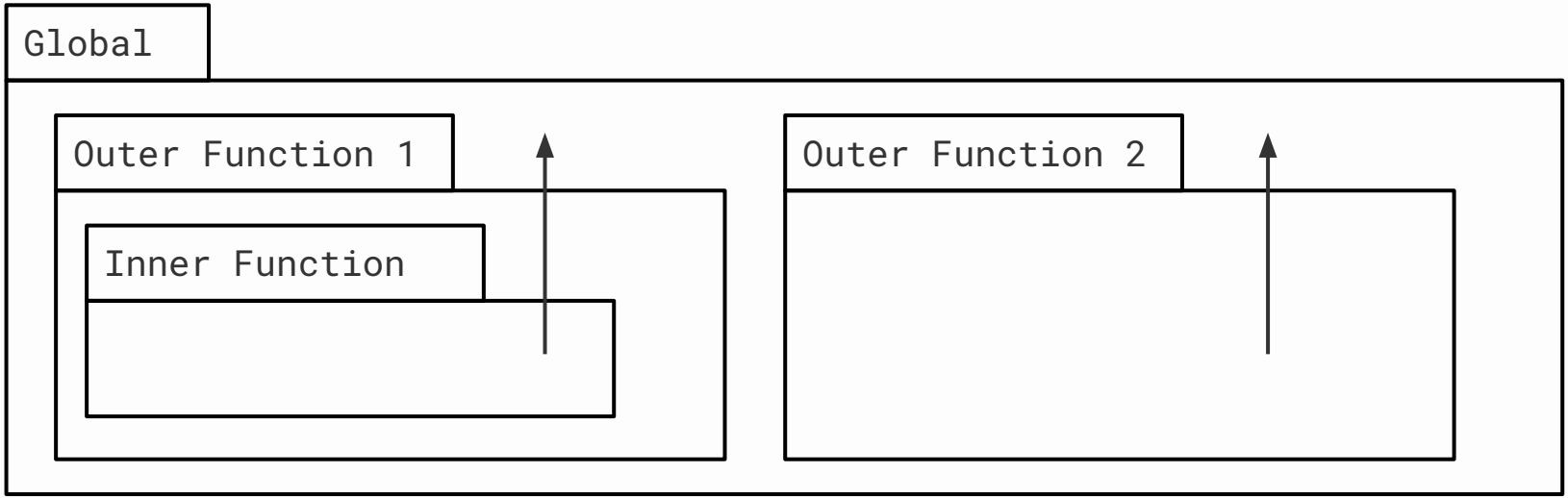
```
1  def double(number):
2      # Define the code here
3
4  x = 3
5  print(double(x))
```

Function Scope

Understanding namespaces

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)
```

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)
```

But functions can’t write outside

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

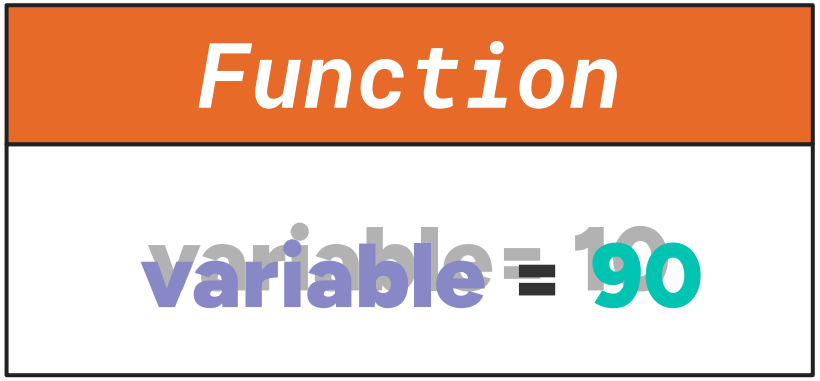
```
x = 10
def function(x):
    print(“Inner”, x)
    x = 5
    print(“Inner”, x)

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

Overwrite using Return

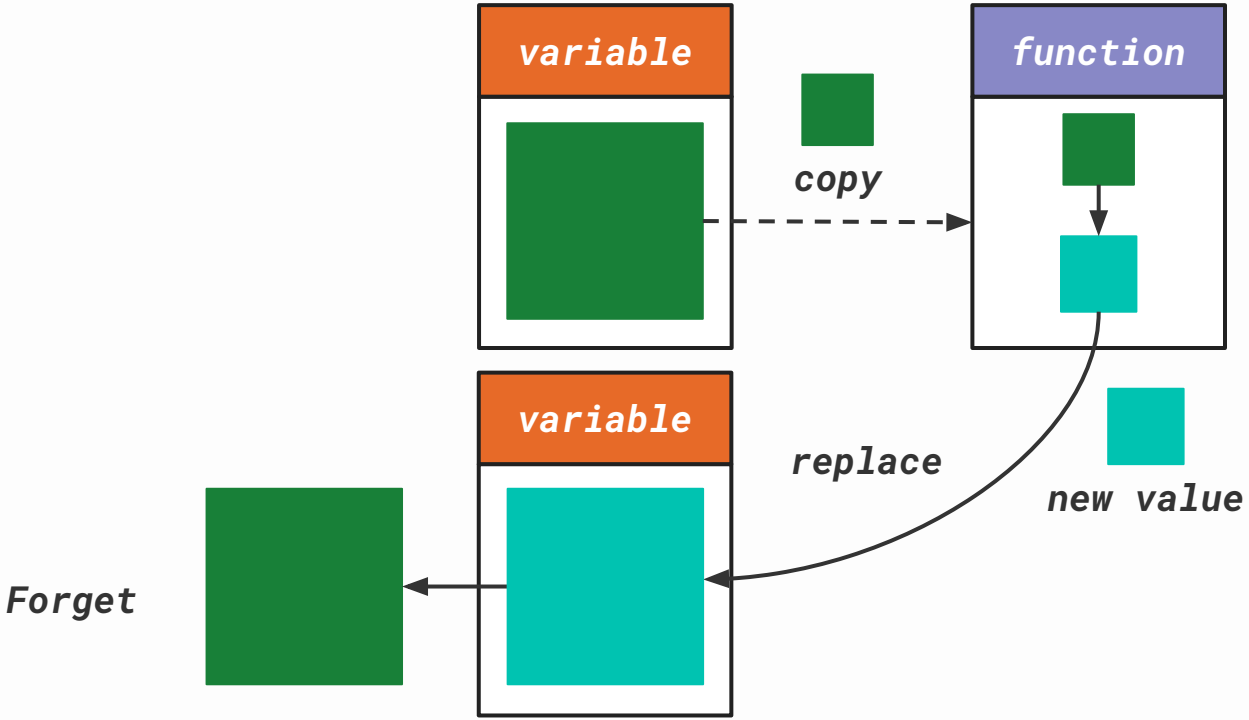
Even if the variable is given as an input, this does not change anything

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

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

variable= 10



variable = function (variable)



Function +

Extra, built-in features for functions

Lambda Functions

Lambda functions are small, anonymous functions defined for simple return processes

lambda args : return_expression

```
1  def triple(x):
2      return (x*3)
3  print(triple(4))
```

```
1  triple = lambda x: x*3
2  print(triple(4))
```

Lambda Functions (Multiple Inputs)

Lambda functions are small, anonymous functions defined for simple return processes

lambda args : return_expression

```
1  def product(x, y):
2      return x*y
3  print(product(2, 4))
```

```
1  product = lambda x, y: x*y
2  print(product(2, 4))
```

Quick Exercise: Lambda Conversion

Convert the following regular function into a lambda function

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

Test the function by calculating the following values

```
4  first_distance = distance(3, 4)
5  second_distance = distance(6, 8)
```

Is using a lambda preferable for this case?

Map Function

The Map function applies a given function to every item in a collection of item (like a list)

```
1  def squared(x):
2      return x**2
```

```
4  numbers = [1, 2, 3, 4, 5]
5  numbers_squared = map(squared, numbers)
6  print(list(numbers_squared))
```

Map Function (with Lambdas)

The Map function applies a given function to every item in a collection of item (like a list)

```
1 numbers = [1, 2, 3, 4, 5]
2 numbers_squared = map(lambda x: x**2, numbers)
3 print(list(numbers_squared))
```

Quick Exercise: Cost Cutting

Divide every item in the given `cost` by two

```
1 cost = [10_000, 200, 31, 45, 1]
2 cost_half = ...
3 print(cost_half)
```

Filter Function

The Filter function keeps the items in a collection of item (like a list) if satisfies a function

```
1  def is_even(x):
2      return x % 2 == 0
```

```
4  numbers = [1, 2, 3, 4, 5]
5  numbers_even = filter(is_even, numbers)
6  print(list(numbers_even))
```

Filter Function (with Lambdas)

The Filter function keeps the items in a collection of item (like a list) if satisfies a function

```
1 numbers = [1, 2, 3, 4, 5]
2 numbers_even = filter(lambda x: x % 2 == 0, numbers)
3 print(list(numbers_squared))
```

Quick Exercise: Top Performers

Given the following `scores`, keep the `scores` that are greater than 6

```
1 scores = [10, 7,5, 3, 5, 8]
2 scores_top = ...
3 print(scores_top)
```

Decorator

A decorator is a function that modifies another function without changing its code

```
1  def decorator(function):
2      def wrapper():
3          print("Before the function runs...")
4          function()
5          print("After the function runs...")
6      return wrapper
```

```
7  @decorator
8  def say_hello():
9       print("Hello World")
10
11 say_hello()
```

Decorator for Function with Inputs

```
1  def decorator(function):
2      def wrapper(user):
3          if user != "admin":
4              print("Access denied!")
5          else:
6              return function(user)
7      return wrapper
```

```
8  @decorator
9  def access_database(user):
10     print("Accessing database...")
11
12 access_database(“user”)
13 access_database(“admin”)
```

F3

Code Organize

Examples of when to use functions

Sum of Squares

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total_squares = 0
for num in numbers:
    if num % 2 == 0:
        total_squares += num ** 2
print(sum(squares))
```

Factorial Calculation

```
1  n = int(input())
2  f = 1
3  for i in range(1, n + 1):
4      f *= i
5  print(f)
```

Hands-On: Reimbursement

Good news! Your company now allows the reimbursement of more than five items.

```
item_count = Input number of items to reimburse
```

Based on the item_count, continually ask for item values

```
Input first item price here
Input second item price here
Input third item price here
...
```

The reimbursement system has changed. It starts at 100%, then 90%, 80%, until it reaches 0%

H3

Refactor Exercise

Practice organizing code using functions

Number Classification

```
n = int(input())
if n % 2 == 0:
    print("Even")
else:
    print("Odd")

n = int(input())
if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Average Grade

```
1  name = input("Name: ")
2  midterm = float(input("Midterm grade: "))
3  final = float(input("Final grade: "))
4  average = (midterm + final) / 2
5
6  if average >= 75:
7      result = "Passed"
8  else:
9      result = "Failed"
10
11 print(name)
12 print("Average:", average)
13 print("Result:", result)
```

Pay Rate

```
1  hours = float(input())
2  rate = float(input())
3  print("Pay:", hours * rate)
4
5  hours = float(input())
6  rate = float(input())
7  print("Pay:", hours * rate)
```

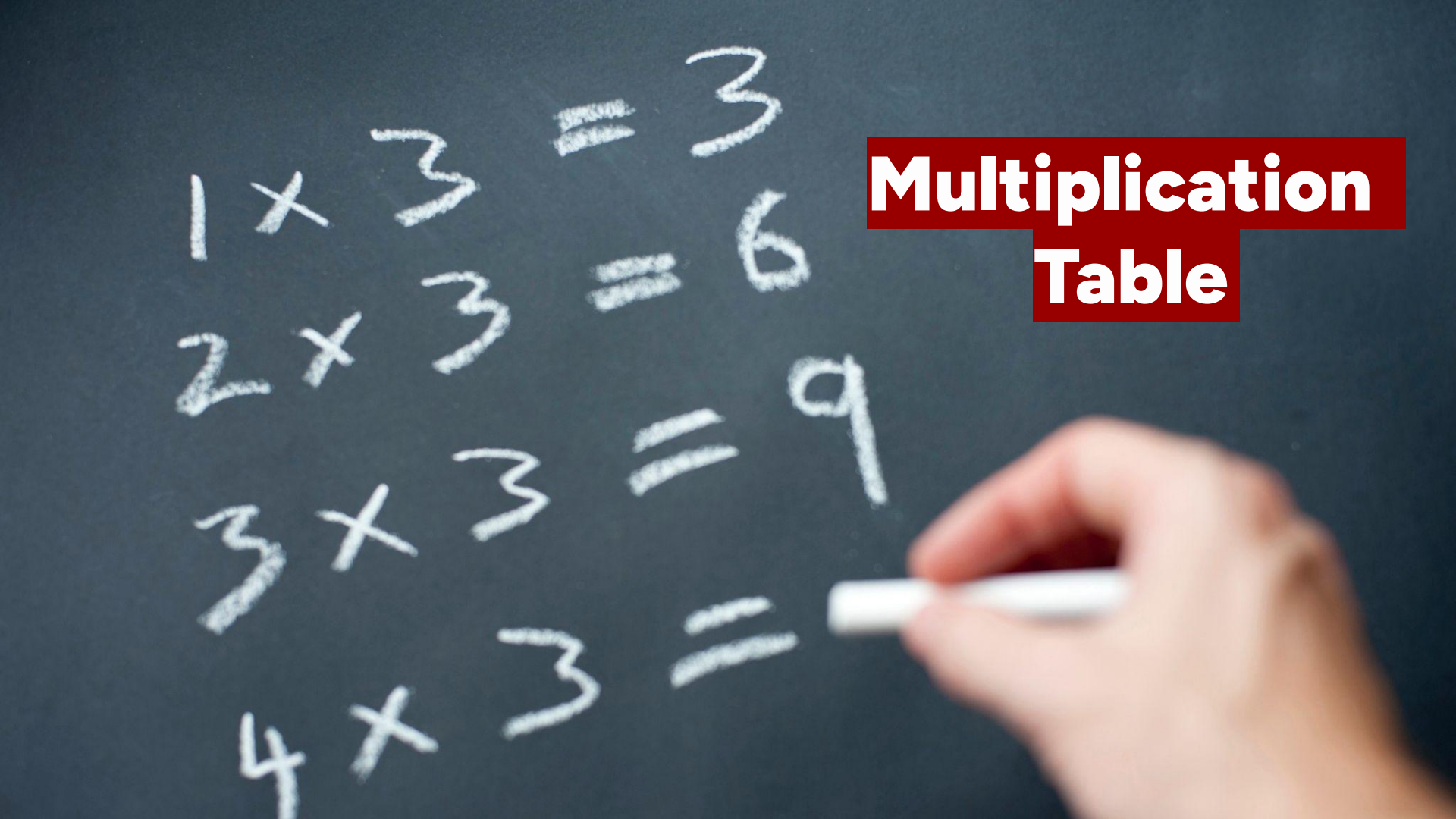
Temperature Conversion

```
1  c = int(input())
2  f = (c * 9 / 5) + 32
3  print(f)
4
5  c = int(input())
6  f = (c * 9 / 5) + 32
7  print(f)
```

05

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

Data Structures

Grouped data

02

Packaging

Handling Python Files

03

String+

Format & Methods

04

File Manage

Persistent Storage

05

Lab Session

Culminating Exercise

Python: Day 01

Introduction and Basic Syntax