DEERWALK TRAINING CENTRE

PYTHON PROGRAM DETAILED GUIDE

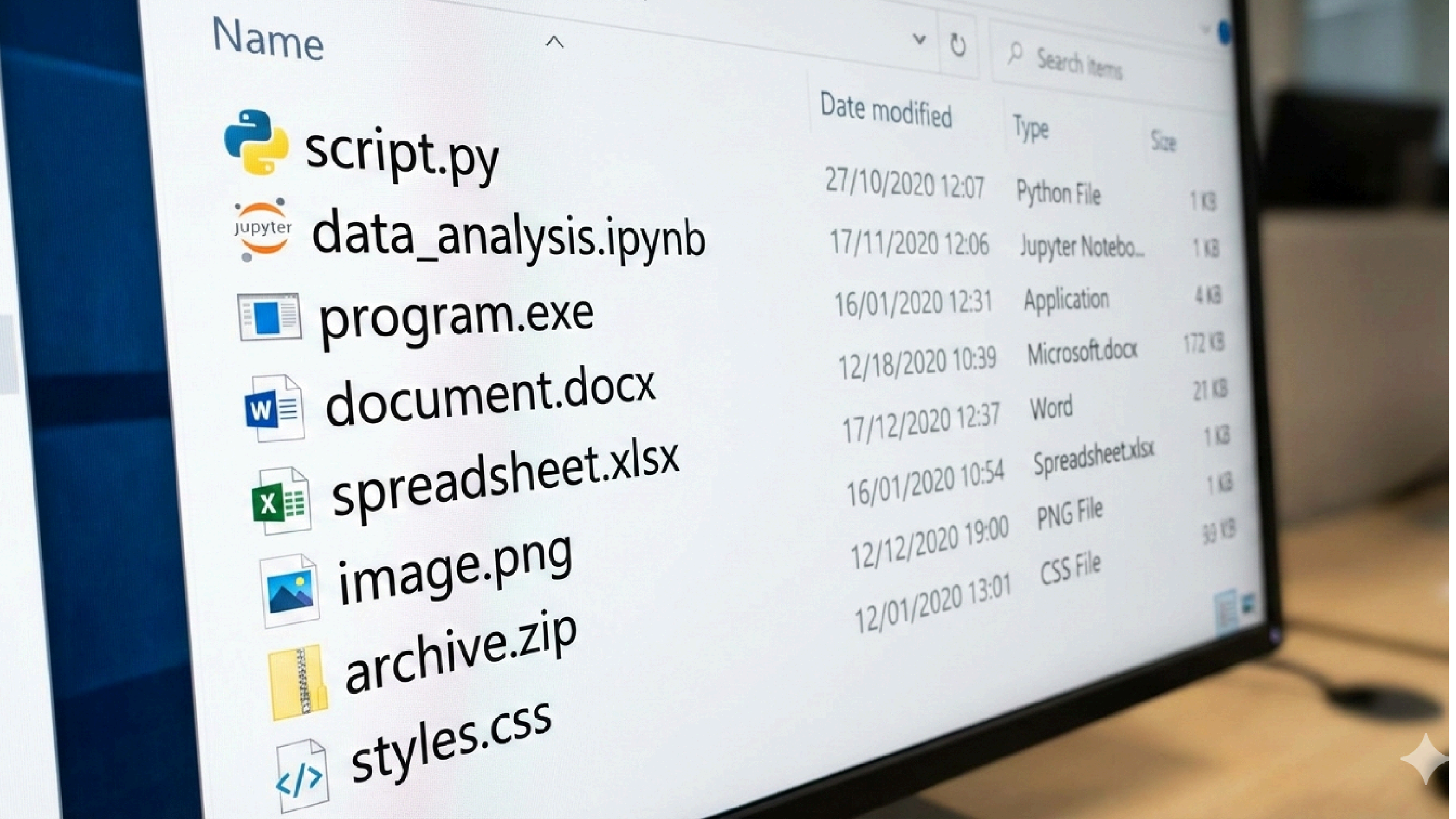
By Ananda Rimal

python introduction

Compiler	Interpreter
A compiler translate the compiler source program in a single line	An interpreter translate the source program line by line
It is faster	It is slower
It consumes less time	It consumes more time them compiler
It is more efficient	It is less efficient
Compilers are larger in size	Interpreters are smaller than compilers

How to install python

1.downalod the python version from the official website
2.install the vs code or any code editor
3.setup python extension with vs code
4.finally run the program


Name
Search items
Date modified
Type
Size
script.py
27/10/2020 12:07
Python File
1 KB
data_analysis.ipynb
17/11/2020 12:06
Jupyter Notebo...
1 KB
program.exe
16/01/2020 12:31
Application
4 KB
document.docx
12/18/2020 10:39
Microsoft.docx
172 KB
spreadsheet.xlsx
17/12/2020 12:37
Word
21 KB
image.png
16/01/2020 10:54
Spreadsheet.xlsx
1 KB
archive.zip
12/12/2020 19:00
PNG File
styles.css
12/01/2020 13:01
CSS File

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

Python Indentation

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

print()-Function

You can use the print() function as many times as you want.
Each call prints text on a new line by default:

Example

```
print("Hello World!")
print("I am learning Python.")
print("It is awesome!")
```

Text in Python must be inside quotes. You can use either " double quotes or ' single quotes:

'hello'

"Hello"

Example

```
print(This will cause an error)
```

Result:

```
SyntaxError: invalid syntax.
```

```
print(3)
print(358)
print(50000)
```

Example

```
print(3 + 3)
print(2 * 5)
```

Mix text and Numbers

Example

```
print("I am", 35, "years old.")
```

Comment

```
#This is a comment
print("Hello, World!")
```

Example

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

Example

Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Example

Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Example

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

Example

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

output ??

Exercise ?

Consider the following code:

```
print('Hello', 'World')
```

What will be the printed result?

- Hello, World
- Hello World
- HelloWorld

What is a correct way to declare a Python variable?

- `var x = 5`
- `#x = 5`
- `$x = 5`
- `x = 5`

True or False:

You can declare string variables with single or double quotes.

```
x = "John"
# is the same as
x = 'John'
```

True or False:

Variable names are not case-sensitive.

```
a = 5
# is the same as
A = 5
```

- ○ True
- ○ False

Select the correct functions to print the data type of a variable:

```
____(____(myvar))
```

typ type var print echo

Which is NOT a legal variable name?

- `my-var = 20`
- `my_var = 20`
- `Myvar = 20`
- `_myvar = 20`

Create a variable named `carname` and assign the value `Volvo` to it.

=

Show Answer

Create a variable named `x` and assign the value `50` to it.

```
 = 
```

Show Answer

What is a correct syntax to add the value 'Hello World', to 3 variables in one statement?

- `x, y, z = 'Hello World'`
- `x = y = z = 'Hello World'`
- `x|y|z = 'Hello World'`

Insert the correct syntax to assign values to multiple variables in one line:

```
x  y  z = "Orange", "Banana", "Cherry"
```

Show Answer

Consider the following code:

```
fruits = ['apple', 'banana', 'cherry']
a, b, c = fruits
print(a)
```

What will be the result of `a`

- apple
- banana
- cherry

Consider the following code:

```
print('Hello', 'World')
```

What will be the printed result?

- Hello, World
- Hello World
- HelloWorld

Consider the following code:

```
a = 'Hello'
b = 'World'
print(a + b)
```

What will be the printed result?

- a + b
- Hello World
- HelloWorld

Consider the following code:

```
a = 4
b = 5
print(a + b)
```

What will be the printed result?

- 45
- 9
- 4 + 5

Python Data Structure

`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple
`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool

Typecasting

```
x = int(1)    # x will be 1
y = int(2.8)  # y will be 2
z = int("3")  # z will be 3
```

Example

Floats:

```
x = float(1)      # x will be 1.0
y = float(2.8)    # y will be 2.8
z = float("3")    # z will be 3.0
w = float("4.2") # w will be 4.2
```

```
print("It's alright")
print("He is called 'Johnny'")
print('He is called "Johnny"')
```

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

python string modify

1.upper()
2.lower()
3.strip() # remove whitespace

Replace String

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Replace String

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

What is a correct syntax to print a string in upper case letters?

- `'Welcome'.upper()`
- `'Welcome'.toUpper()`
- `'Welcome'.toUpperCase()`

Return the string without any whitespace at the beginning or the end.

```
txt = " Hello World "
x = 
```

Boolean Values

In programming you often need to know if an expression is `True` or `False`.

You can evaluate any expression in Python, and get one of two answers, `True` or `False`.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python List

```
mylist = ["apple", "banana", "cherry"]
```

Beginner Level (Basics of Lists)

1. Create a list of 10 numbers and print it
2. Find the sum and average of list elements
3. Find the maximum and minimum element
4. Reverse a list
5. Remove duplicate elements from a list
6. Find the length of a list
7. Sort a list in ascending order
8. Copy one list into another

List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Negative Indexing

Negative indexing means start from the end

`-1` refers to the last item, `-2` refers to the second last item etc.

Example

Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Change Item Value

To change the value of a specific item, refer to the index number:

Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Append Items

To add an item to the end of the list, use the `append()` method:

Example

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Insert Items

To insert a list item at a specified index, use the `insert()` method.

The `insert()` method inserts an item at the specified index:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Extend List

To append elements from *another list* to the current list, use the `extend()` method.

Example

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

Remove Specified Item

The `remove()` method removes the specified item.

Example

Remove "banana":

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

If there are more than one item with the specified value, the `remove()` method removes the first occurrence:

Example

Remove the first occurrence of "banana":

```
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

Remove Specified Index

The `pop()` method removes the specified index.

Example

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

If you do not specify the index, the `pop()` method removes the last item.

Example

Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Clear the List

The `clear()` method empties the list.

The list still remains, but it has no content.

Example

Clear the list content:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Example

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Example

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

Sort Descending

To sort descending, use the keyword argument `reverse = True`:

Example

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Example

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

Use the copy() method

You can use the built-in List method `copy()` to copy a list.

Example

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Use the list() method

Another way to make a copy is to use the built-in method `list()`.

Example

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the `+` operator.

Example

Join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

Another way to join two lists is by appending all the items from list2 into list1, one by one:

Example

Append list2 into list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

Or you can use the `extend()` method, where the purpose is to add elements from one list to another list:

Example

Use the `extend()` method to add list2 at the end of list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

Python has a set of built-in methods that you can use on lists.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

What will be the result of the following syntax:

```
mylist = ['apple', 'banana', 'cherry']
print(mylist[1])
```

- apple
- banana
- cherry

What will be the result of the following syntax:

```
mylist = ['apple', 'banana', 'banana', 'cherry']
print(mylist[2])
```

- apple
- banana
- cherry

True or False.

List items cannot be removed after the list has been created.

- True
- False

Select the correct function for returning the number of items in a list:

```
thislist = ['apple', 'banana', 'cherry']
print(          (thislist))
```

`length` `size` `len` `items`

What will be the result of the following syntax:

```
mylist = ['apple', 'banana', 'cherry']
print(mylist[-1])
```

- apple
- banana
- cherry

Print the second item in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
print(          )
```

What will be the result of the following syntax:

```
mylist = ['apple', 'banana', 'cherry']
mylist[0] = 'kiwi'
print(mylist[1])
```

- apple
- banana
- cherry
- kiwi

Change the value from "apple" to "kiwi", in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
[redacted] = [redacted]
```

What will be the result of the following syntax:

```
mylist = ['apple', 'banana', 'cherry']
mylist[1:2] = ['kiwi', 'mango']
print(mylist[2])
```

- banana
- cherry
- kiwi
- mango

What will be the result of the following syntax:

```
mylist = ['apple', 'banana', 'cherry']
mylist.insert(0, 'orange')
print(mylist[1])
```

- apple
- banana
- cherry
- orange

Use the `append` method to add "orange" to the `fruits` list.

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

Use the `insert` method to add "lemon" as the second item in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
                    "lemon")
```

```
mytuple = ("apple", "banana", "cherry")
```

Tuple

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

Tuple Length

To determine how many items a tuple has, use the `len()` function:

Example

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example

One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Tuple Items - Data Types

Tuple items can be of any data type:

Example

String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

A tuple can contain different data types:

Example

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```

Example

What is the data type of a tuple?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary Items

Dictionary items are ordered, changeable, and do not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

Example

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Example

Make a change in the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Removing Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Example

Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

The Python Structural Hierarchy

Concept	What is it?	Scope	Control (IoC*)	Real-World Analogy
Module	A single `.py` file.	Smallest unit of code.	**You** call it.	A single tool (like a hammer).
Package	A folder containing multiple modules.	Medium; organizes related files.	**You** call it.	A toolbox of related tools.
Library	A collection of packages for a specific task.	Large; solves a domain problem.	**You** call it.	A specialized workshop (e.g., plumbing).
Framework	A skeleton/architecture for an entire app.	Comprehensive; provides the "rules."	**It** calls you.	The entire factory assembly line.

Random Module

lucky draw game
random people choice game
love calculator

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

Example

If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

How If Statements Work

The if statement evaluates a condition (an expression that results in `True` or `False`). If the condition is true, the code block inside the if statement is executed. If the condition is false, the code block is skipped.

Example

Checking if a number is positive:

```
number = 15
if number > 0:
  print("The number is positive")
```

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Multiple Statements in If Block

You can have multiple statements inside an if block. All statements must be indented at the same level.

Example

Multiple statements in an if block:

```
age = 20
if age >= 18:
  print("You are an adult")
  print("You can vote")
  print("You have full legal rights")
```

Using Variables in Conditions

Boolean variables can be used directly in if statements without comparison operators.

Example

Using a boolean variable:

```
is_logged_in = True
if is_logged_in:
  print("Welcome back!")
```

The Elif Keyword

The `elif` keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

The `elif` keyword allows you to check multiple expressions for `True` and execute a block of code as soon as one of the conditions evaluates to `True`.

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Multiple Elif Statements

You can have as many `elif` statements as you need. Python will check each condition in order and execute the first one that is true.

Example

Testing multiple conditions:

```
score = 75

if score >= 90:
  print("Grade: A")
elif score >= 80:
  print("Grade: B")
elif score >= 70:
  print("Grade: C")
elif score >= 60:
  print("Grade: D")
```

Day of the week checker:

```
day = 3

if day == 1:
  print("Monday")
elif day == 2:
  print("Tuesday")
elif day == 3:
  print("Wednesday")
elif day == 4:
  print("Thursday")
elif day == 5:
  print("Friday")
elif day == 6:
  print("Saturday")
elif day == 7:
  print("Sunday")
```

The Else Keyword

The `else` keyword catches anything which isn't caught by the preceding conditions.

The `else` statement is executed when the `if` condition (and any `elif` conditions) evaluate to `False`.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Else Without Elif

You can also have an `else` without the `elif`:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Example

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Python Logical Operators

Logical operators are used to combine conditional statements. Python has three logical operators:

- `and` - Returns True if both statements are true
- `or` - Returns True if one of the statements is true
- `not` - Reverses the result, returns False if the result is true

The or Operator

The `or` keyword is a logical operator, and is used to combine conditional statements. At least one condition must be true for the entire expression to be true.

Example

Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33
b = 200


if b > a:
  pass
```

Why Use pass?

The `pass` statement is useful in several situations:

- When you're creating code structure but haven't implemented the logic yet
- When a statement is required syntactically but no action is needed
- As a placeholder for future code during development
- In empty functions or classes that you plan to implement later

---

pass in Development

During development, you might want to sketch out your program structure before implementing the details. The `pass` statement allows you to do this without syntax errors.

Example

Placeholder for future implementation:

```
age = 16

if age < 18:
  pass # TODO: Add underage logic later
else:
  print("Access granted")
```

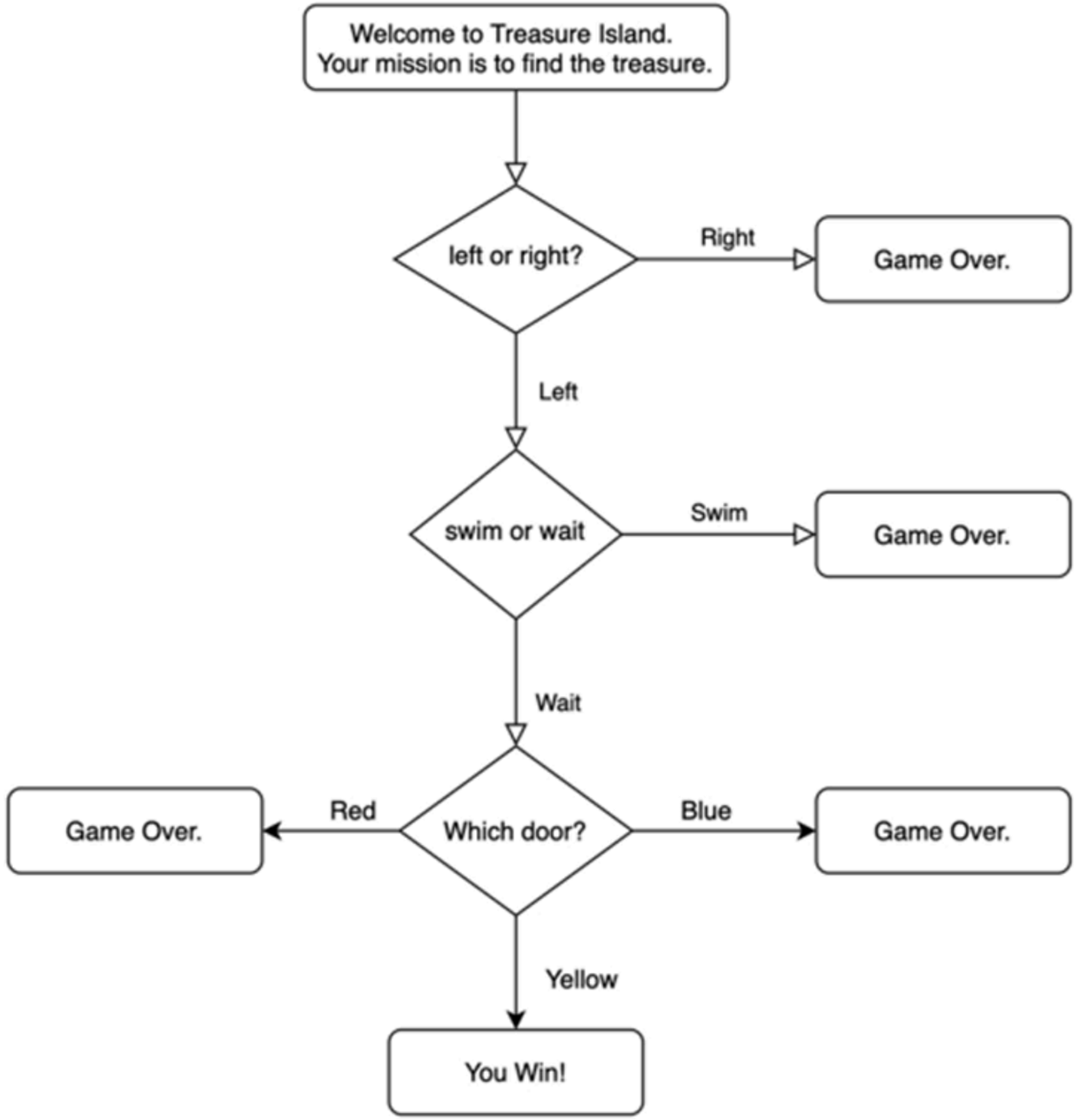
Nested If Statements

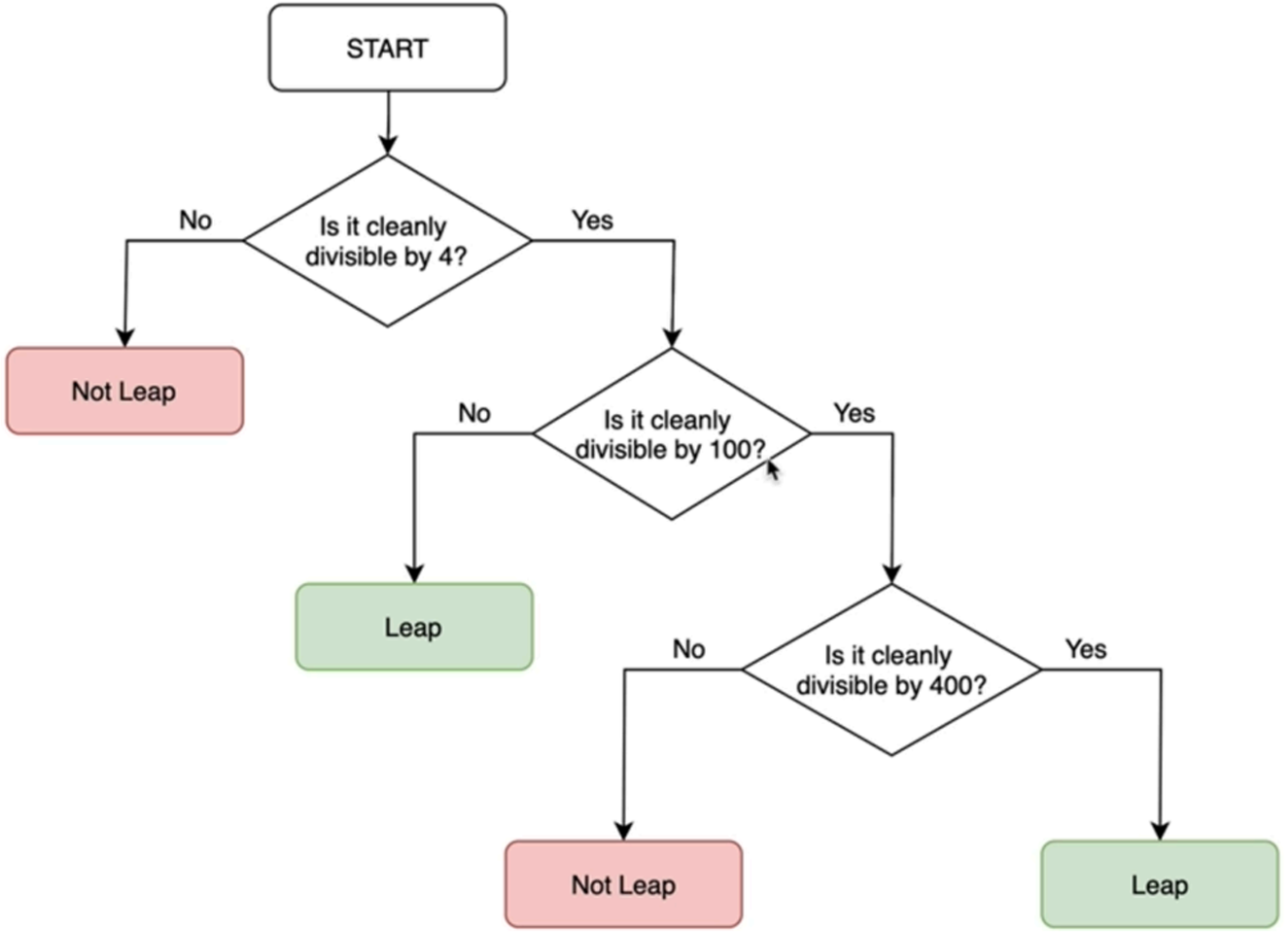
You can have `if` statements inside `if` statements. This is called *nested* `if` statements.

Example

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

BMI PROGRAM

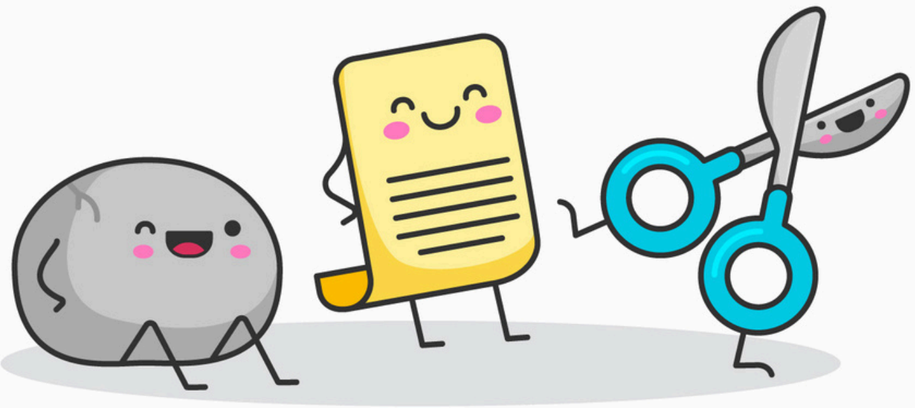

Welcome to Treasure Island.
Your mission is to find the treasure.
left or right?
Right
Game Over.
Left
swim or wait
Swim
Game Over.
Wait
Which door?
Red
Game Over.
Blue
Game Over.
Yellow
You Win!


START
Is it cleanly divisible by 4?
No
Yes
Not Leap
Is it cleanly divisible by 100?
No
Yes
Leap
Is it cleanly divisible by 400?
No
Yes
Not Leap
Leap

odd even program

prime or not

your task



ROCK • PAPER • SCISSORS

Now make a detailed note on notebook about variables data structure and if else

loops

Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when `x` is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):
  print(x)
```

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):
  print(x)
```

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
  print(x)
```

🟢 Beginner Level

🔁 `for` loop questions

1. Print numbers from 1 to 10
2. Print even numbers from 1 to 20
3. Print odd numbers from 1 to 20
4. Print all elements of a list
5. Print each character of a string
6. Print multiplication table of 5
7. Print squares of numbers from 1 to 10
8. Print numbers in reverse order (10 to 1)
9. Print all vowels in a string
10. Print sum of numbers from 1 to 10

🔄 `while` loop questions

1. Print numbers from 1 to 10
2. Print numbers from 10 to 1
3. Print even numbers up to 50
4. Print sum of first 10 numbers
5. Print multiplication table of 3
6. Count digits in a number
7. Reverse a number
8. Print squares of numbers from 1 to 10
9. Print all digits of a number
10. Print factorial of a number

Python Functions

A function is a block of code which only runs when it is called.

A function can return data as a result.

A function helps avoiding code repetition.

Creating a Function

In Python, a function is defined using the `def` keyword, followed by a function name and parentheses:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, write its name followed by parentheses:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

python return vs print

calculator program

oop
object oriented programming

What is OOP?

OOP stands for **Object-Oriented Programming**.

Python is an object-oriented language, allowing you to structure your code using classes and objects for better organization and reusability.

Advantages of OOP

- Provides a clear structure to programs
- Makes code easier to maintain, reuse, and debug
- Helps keep your code DRY (**Don't Repeat Yourself**)
- Allows you to build reusable applications with less code

Tip: The DRY principle means you should avoid writing the same code more than once. Move repeated code into functions or classes and reuse it.

What are Classes and Objects?

Classes and objects are the two core concepts in object-oriented programming.

A class defines what an object should look like, and an object is created based on that class. For example:

Class	Objects
Fruit	Apple, Banana, Mango
Car	Volvo, Audi, Toyota

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

Multiple Objects

You can create multiple objects from the same class:

Example

Create three objects from the MyClass class:

```
p1 = MyClass()
p2 = MyClass()
p3 = MyClass()

print(p1.x)
print(p2.x)
print(p3.x)
```

The __init__() Method

All classes have a built-in method called `__init__()`, which is always executed when the class is being initiated.

The `__init__()` method is used to assign values to object properties, or to perform operations that are necessary when the object is being created.

Example

Create a class named Person, use the `__init__()` method to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Emil", 36)

print(p1.name)
print(p1.age)
```

Why Use __init__()?

Without the `__init__()` method, you would need to set properties manually for each object:

Example

Create a class without `__init__()`:

```
class Person:
  pass

p1 = Person()
p1.name = "Tobias"
p1.age = 25

print(p1.name)
print(p1.age)
```

Using `__init__()` makes it easier to create objects with initial values:

Example

With `__init__()`, you can set initial values when creating the object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Linus", 28)

print(p1.name)
print(p1.age)
```

Default Values in __init__()

You can also set default values for parameters in the `__init__()` method:

Example

Set a default value for the age parameter:

```
class Person:
  def __init__(self, name, age=18):
    self.name = name
    self.age = age

p1 = Person("Emil")
p2 = Person("Tobias", 25)

print(p1.name, p1.age)
print(p2.name, p2.age)
```

Multiple Parameters

The `__init__()` method can have as many parameters as you need:

Example

Create a Person class with multiple parameters:

```
class Person:
  def __init__(self, name, age, city, country):
    self.name = name
    self.age = age
    self.city = city
    self.country = country

p1 = Person("Linus", 30, "Oslo", "Norway")

print(p1.name)
print(p1.age)
print(p1.city)
print(p1.country)
```

The self Parameter

The `self` parameter is a reference to the current instance of the class.

It is used to access properties and methods that belong to the class.

Example

Use `self` to access class properties:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def greet(self):
    print("Hello, my name is " + self.name)

p1 = Person("Emil", 25)
p1.greet()
```

Why Use self?

Without `self`, Python would not know which object's properties you want to access:

Example

The `self` parameter links the method to the specific object:

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

  def printname(self):
    print(self.name)

p1 = Person("Tobias")
p2 = Person("Linus")

p1.printname()
p2.printname()
```

Access Properties

You can access object properties using dot notation:

Example

Access the properties of an object:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

car1 = Car("Toyota", "Corolla")

print(car1.brand)
print(car1.model)
```

Class Methods

Methods are functions that belong to a class. They define the behavior of objects created from the class.

Example

Create a method in a class:

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

  def greet(self):
    print("Hello, my name is " + self.name)

p1 = Person("Emil")
p1.greet()
```

Methods with Parameters

Methods can accept parameters just like regular functions:

Example

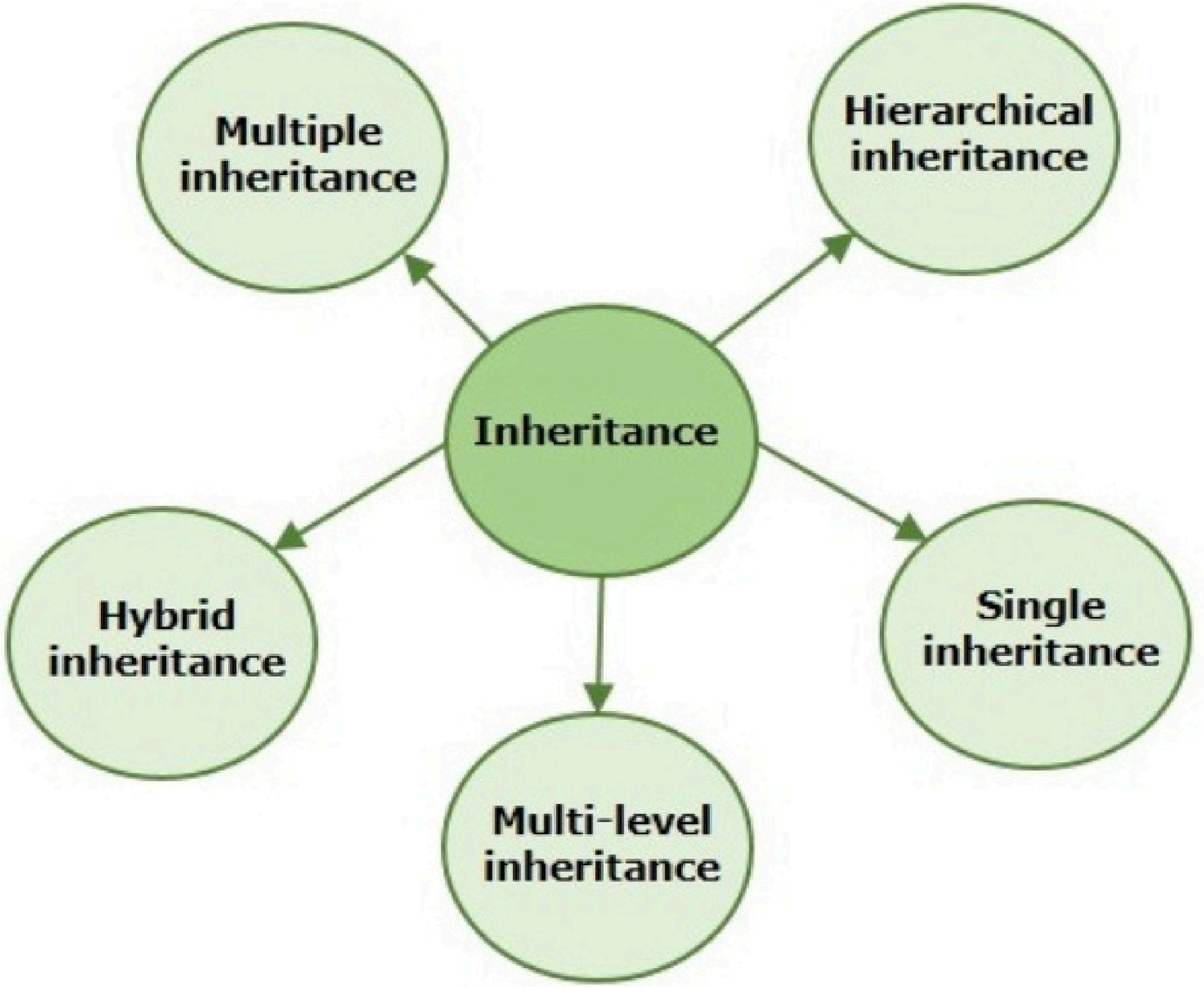
Create a method with parameters:

```
class Calculator:
  def add(self, a, b):
    return a + b

  def multiply(self, a, b):
    return a * b

calc = Calculator()
print(calc.add(5, 3))
print(calc.multiply(4, 7))
```

Example

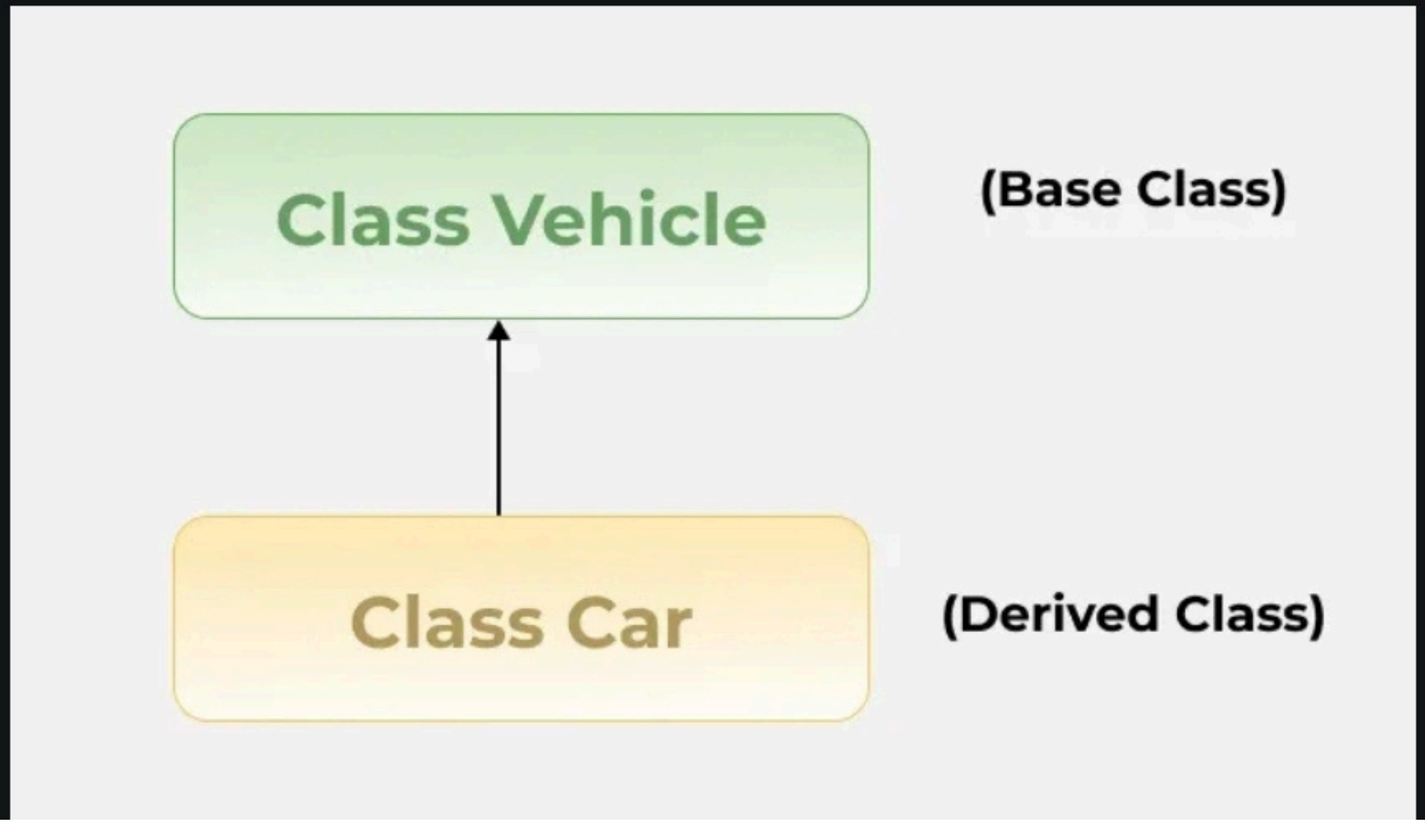
Delete a method from a class:

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

  def greet(self):
    print("Hello!")

p1 = Person("Emil")

del Person.greet

p1.greet() # This will cause an error
```


Multiple inheritance
Hierarchical inheritance
Inheritance
Hybrid inheritance
Single inheritance
Multi-level inheritance

1. Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance.



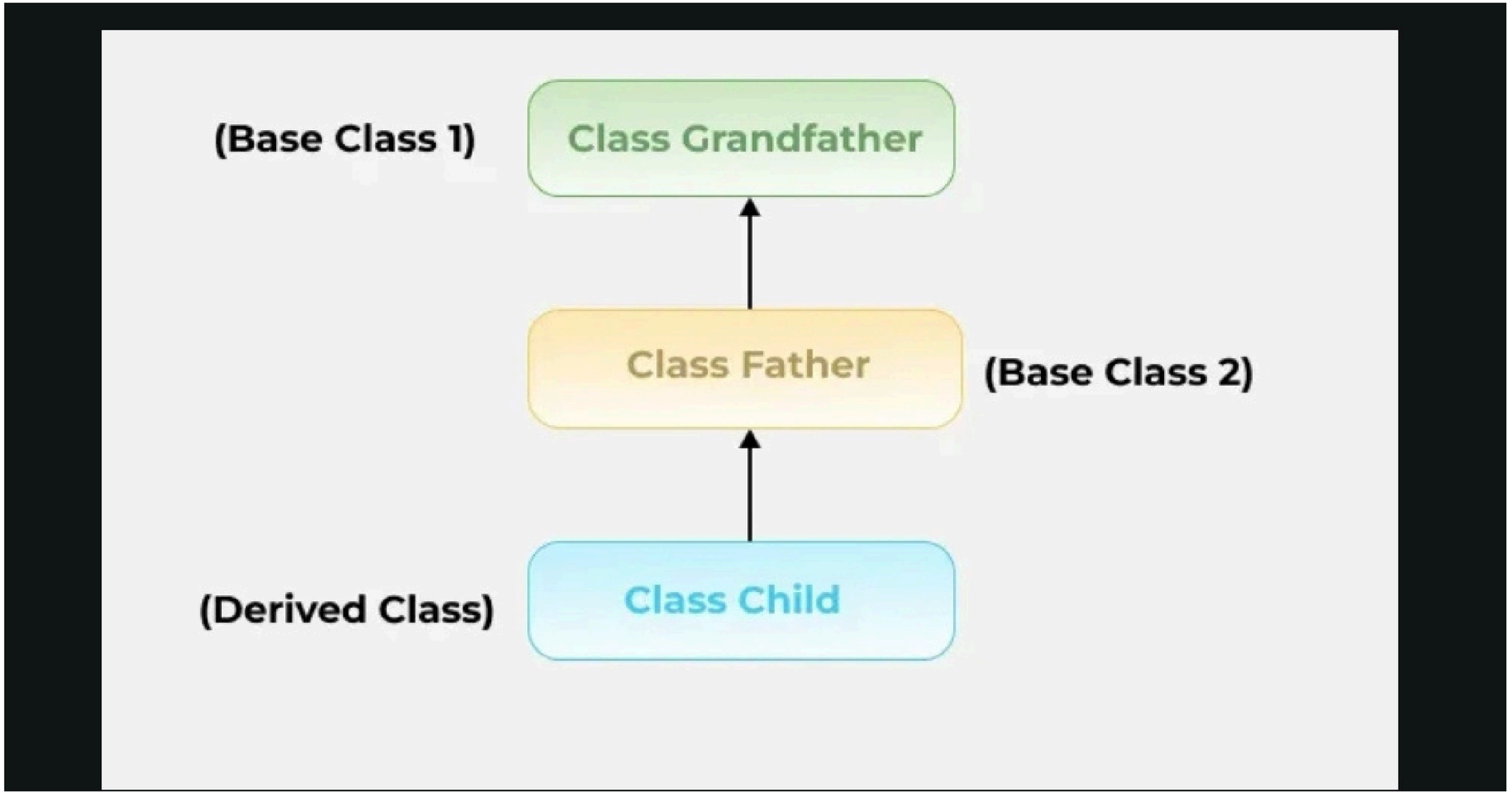
1️⃣ Single Inheritance

📌 **One parent → one child**

python

```python
class Parent:
    def show(self):
        print("Parent class")

class Child(Parent):
    def display(self):
        print("Child class")

c = Child()
c.show()
c.display()
```

Multilevel Inheritance



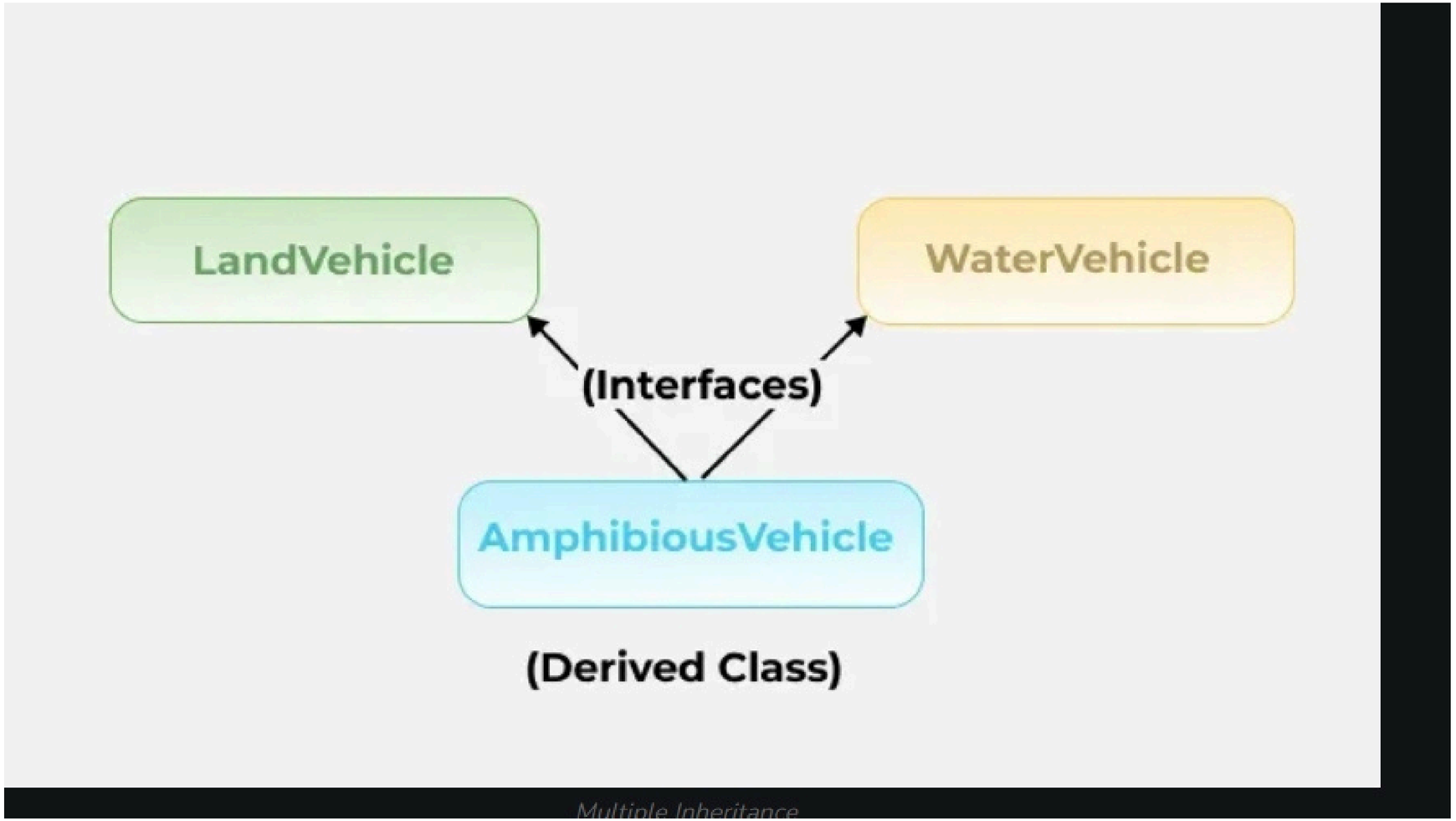
2️⃣ Multilevel Inheritance

📌 **Parent → Child → Grandchild**

python

```python
class GrandParent:
    def house(self):
        print("Grandparent house")

class Parent(GrandParent):
    def car(self):
        print("Parent car")

class Child(Parent):
    def bike(self):
        print("Child bike")

c = Child()
c.house()
c.car()
c.bike()
```

Multiple Inheritance



4️⃣ Multiple Inheritance

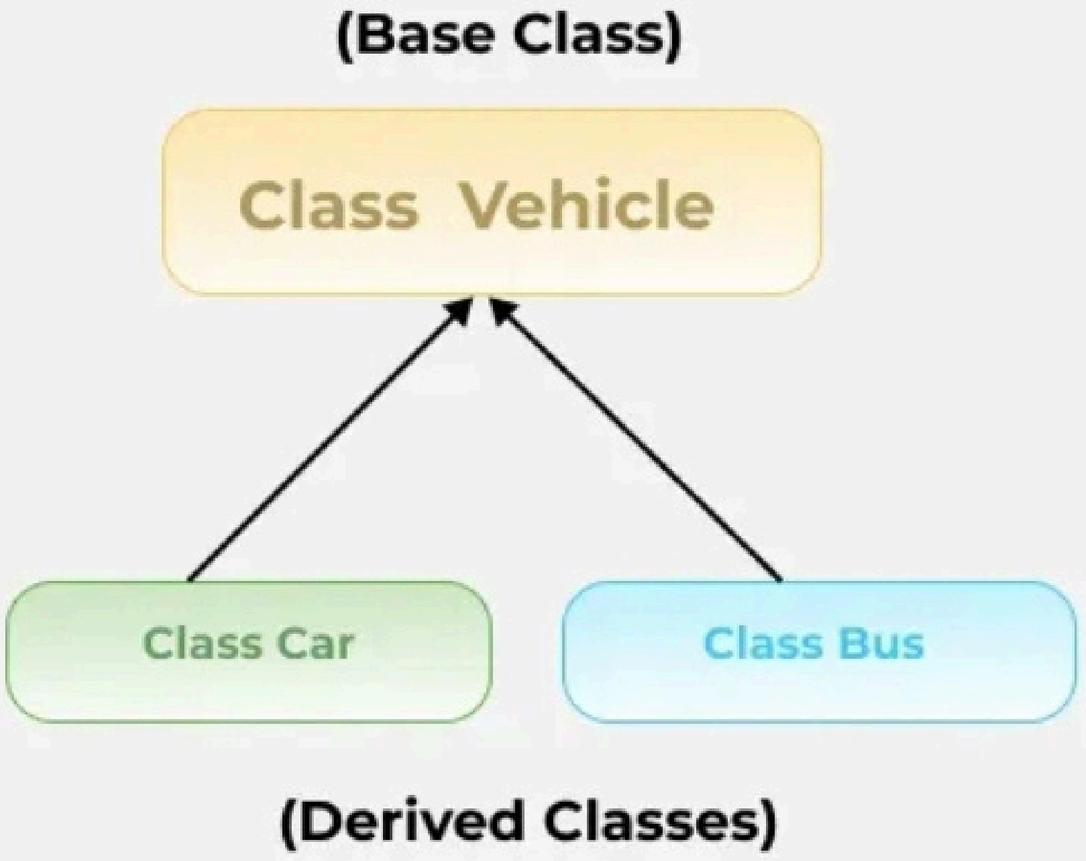
📌 Multiple parents → one child

python

```python
class Father:
    def father_skill(self):
        print("Father skill")

class Mother:
    def mother_skill(self):
        print("Mother skill")

class Child(Father, Mother):
    pass

c = Child()
c.father_skill()
c.mother_skill()
```

Mro

Hierarchical inheritance



3 Hierarchical Inheritance

📌 **One parent → multiple children**

python

```python
class Parent:
    def property(self):
        print("Parent property")

class Child1(Parent):
    pass

class Child2(Parent):
    pass

c1 = Child1()
c2 = Child2()

c1.property()
c2.property()
```

Polymorphism

- Polymorphism means “having more than one form.”
- It allows the same method or operation to behave differently in different situations.
-

General Classification of Polymorphism (OOP Theory)

In **most OOP languages (C++, Java)**, polymorphism is of **two types**:

1. **Static Polymorphism (Compile-Time Polymorphism)**
2. **Dynamic Polymorphism (Run-Time Polymorphism)**

1️⃣ Static Polymorphism (Compile-Time)

📌 **Definition:**

The method call is **resolved at compile time**.

Includes:

- **Method Overloading**
- **Operator Overloading**

Example (Java / C++ idea)

java

```
add(int a, int b)
add(int a, int b, int c)
```

1️⃣ Static Polymorphism (Compile-Time)

📌 **Definition:**

The method call is **resolved at compile time**.

Includes:

- **Method Overloading**
- **Operator Overloading**

Example (Java / C++ idea)

```java
add(int a, int b)
add(int a, int b, int c)
```

- ✔ Same method name
- ✔ Different parameters
- ✔ Decided at compile time

❌ Static Polymorphism in Python

👉 **Python does NOT support compile-time polymorphism**

Why?

- Python is **dynamically typed**
- No compile-time method resolution
- Function names are resolved **at runtime**

So:

> **Traditional method overloading is NOT possible in Python**

❌ Example (Not supported in Python)

python

```python
class Test:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c
```

- Types of Polymorphism in Python
  1. Method Overriding (Dynamic Polymorphism)
  2. Duck Typing

Method Overriding Example (Simple)

python

```python
# Parent class
class Animal:
    def sound(self):
        print("Animal makes a sound")

# Child class
class Dog(Animal):
    def sound(self):
        print("Dog barks")   # overrides parent method

# Using the classes
a = Animal()
a.sound()   # Calls parent method

d = Dog()
d.sound()   # Calls child method (overridden)
```

super Keyword

Python Encapsulation

Encapsulation is about protecting data inside a class.

It means keeping data (properties) and methods together in a class, while controlling how the data can be accessed from outside the class.

This prevents accidental changes to your data and hides the internal details of how your class works.

Private Properties

In Python, you can make properties private by using a double underscore `__` prefix:

Example

Create a private class property named `__age` :

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age # Private property

p1 = Person("Emil", 25)
print(p1.name)
print(p1.__age) # This will cause an error
```

Get Private Property Value

To access a private property, you can create a getter method:

Example

Use a getter method to access a private property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age

  def get_age(self):
    return self.__age

p1 = Person("Tobias", 25)
print(p1.get_age())
```

Get Private Property Value

To access a private property, you can create a getter method:

Example

Use a getter method to access a private property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age

  def get_age(self):
    return self.__age

p1 = Person("Tobias", 25)
print(p1.get_age())
```

Why Use Encapsulation?

Encapsulation provides several benefits:

- **Data Protection:** Prevents accidental modification of data
- **Validation:** You can validate data before setting it
- **Flexibility:** Internal implementation can change without affecting external code
- **Control:** You have full control over how data is accessed and modified

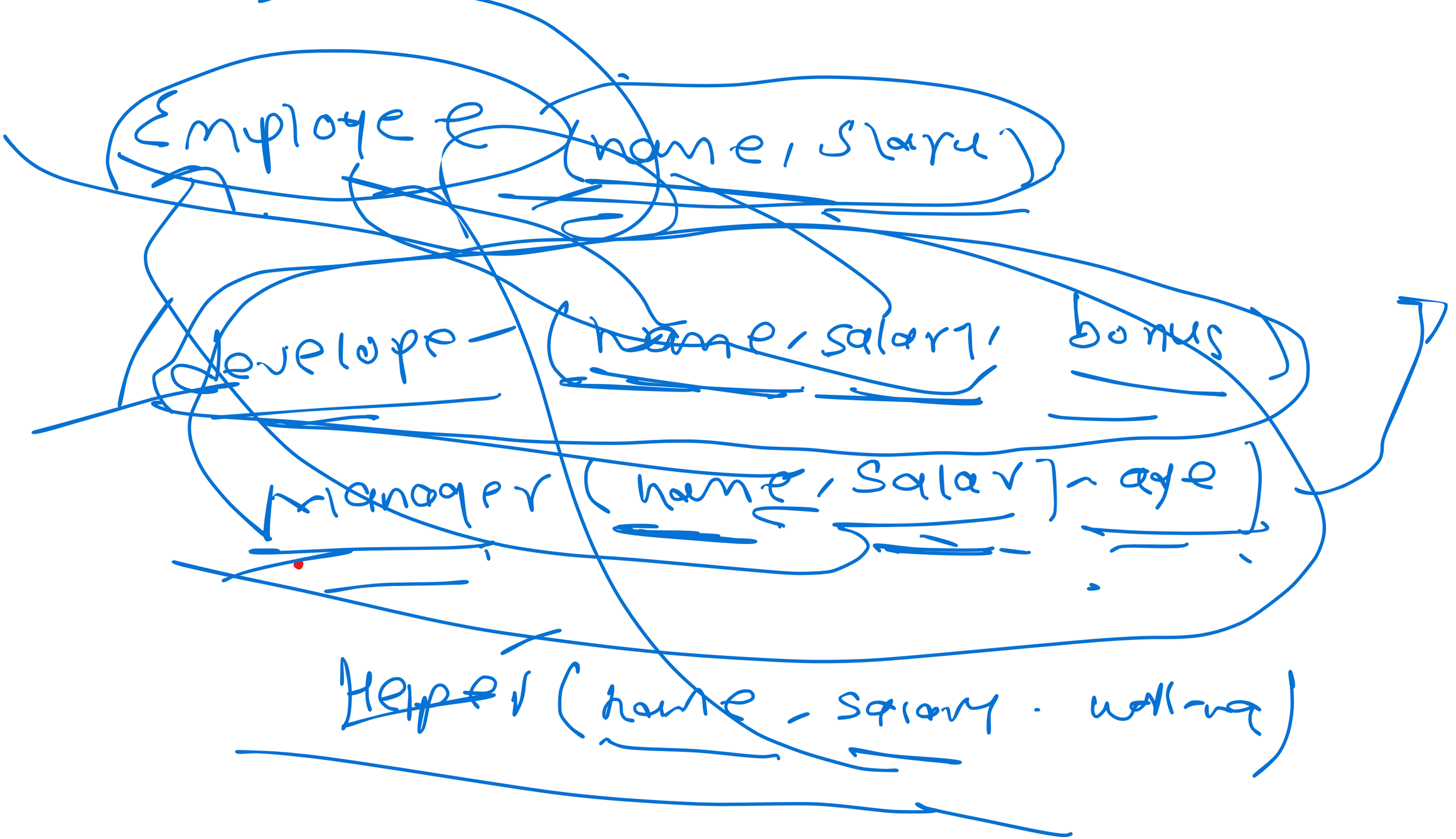
Protected Properties

Python also has a convention for protected properties using a single underscore `_` prefix:

Example

Create a protected property:

```
class Person:
  def __init__(self, name, salary):
    self.name = name
    self._salary = salary # Protected property

p1 = Person("Linus", 50000)
print(p1.name)
print(p1._salary) # Can access, but shouldn't
```


Employee (name, salary)
Developer (name, salary, bonus)
Manager (name, salary) - age
Helper (name, salary, working)