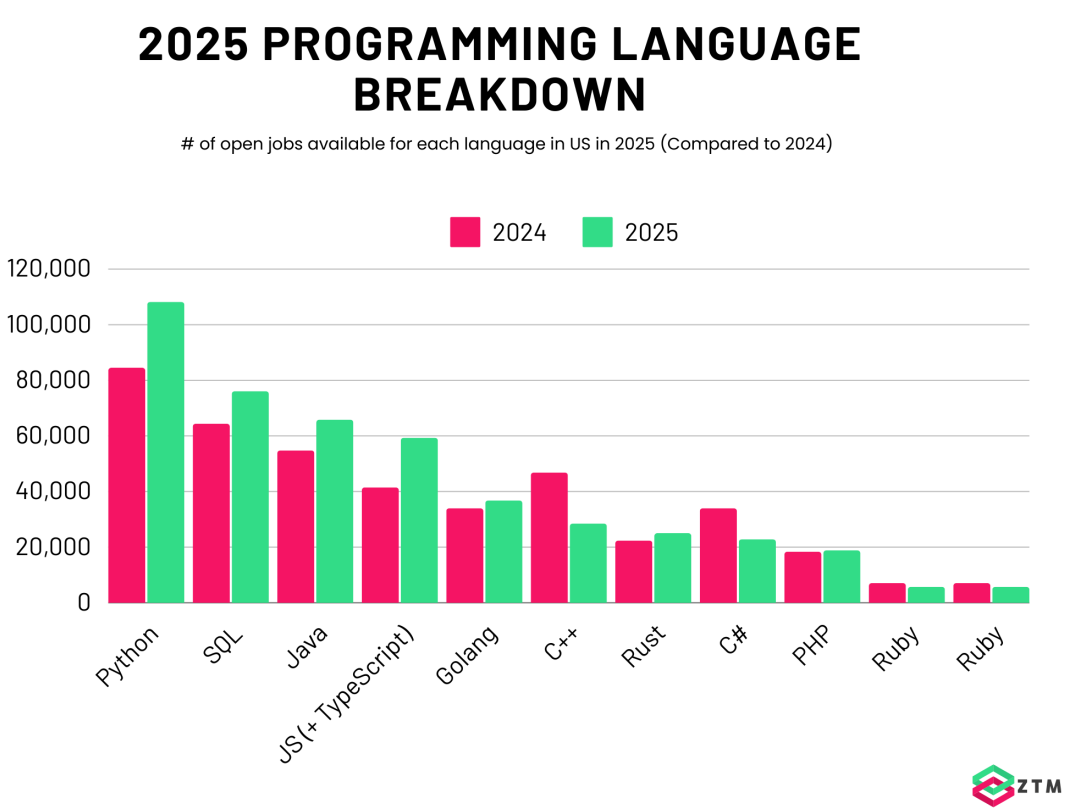
Class 01: Unlocking the Power of Variables and Logic

"

Why Python for Machine Learning?

1. Easy to Learn
2. Versatile
3. Large Community
4. Libraries and Frameworks
5. High Demand for Data Science
6. Automation
7. Cross-platform
8. Big Data

Guido van Rossum



Python Software Foundation: https://www.python.org/psf-landing/

Which Python version we should use?

- Python 3.12 may have compatibility issues.
- Python 3.10 or 3.11 – Best balance of stability, performance, and compatibility for ML.
- Python 2 is deprecated and no longer supported since 2020.

Anaconda Download Link: https://repo.anaconda.com/archive/Anaconda3-2024.02-1-Windows-x86_64.exe

For Other OS: https://repo.anaconda.com/archive/

In [1]:
```
!python --version
```

```
Python 3.11.7
```

Class Topic

1. Variables
2. Data Types
3. Dynamic Types
4. Indentation in Python
5. If Statements
6. Logical Operators
7. Comparison/Relational/Conditional Operators

In [3]:
```
## Your First Python Code
print("Hello World!")
```

```
Hello World!
```

For comments in code use # sign

Variable

Variables act as placeholders for data. They allow us to store and reuse values in our program.

In [14]:
```
# Basic Syntax
a = 5
```

The equal sign (=) is used to assign values to variables.

In [35]:
```
print(type(a))
print(isinstance(5, int))  # Output: True (5 is an instance of int class)
print(isinstance(a, int))  # Output: True
```

```
<class 'int'>
True
True
```

In [15]:
```
# Delete variable a
#del a
print(a)
```

```
5
```

In [32]:
```
#print(dir(int))
```

In [34]:
```
a = 5
b = 5
print(id(a))  # Memory address of object 5
print(id(b))  # Same memory address as a (Python optimizes small integers)
```

```
140709698585512
140709698585512
```

Rules for Naming Variables

To use variables effectively, we must follow Python's naming rules:

1. Variable names can only contain letters, digits and underscores (_).
2. A variable name cannot start with a digit.
3. Variable names are case-sensitive (myVar and myvar are different).
4. Avoid using Python keywords (e.g., if, else, for) as variable names.

In [15]:
```
# and,as,assert
# break,class,continue
# def,del,elif
# else,except,False
# finally,for,from
# global,if,import
# in,is,lambda
# None,nonlocal,not
# or,pass,raise
# return,True,try
# while,with,yield
```

In [18]:
```
## Valid variable name
my_var = 10
_name = 'Taimur'
var123 = 3.14
x, y, z = 1, 2, 3
a,b,c = 1,2,"Zara Ali"
a_b_c = "Python"
PI = 3.1416
a = b = c = 100
```

In [ ]:
```
## Invalid variable name
1var = 5
my-var = 10
class = "test"
my variable = 20
@data = "hello"
def = 100
```

In [25]:
```
# Use meaningful names:
customer_name = "Taimur" # Good
c = "Taimur" # Bad
```

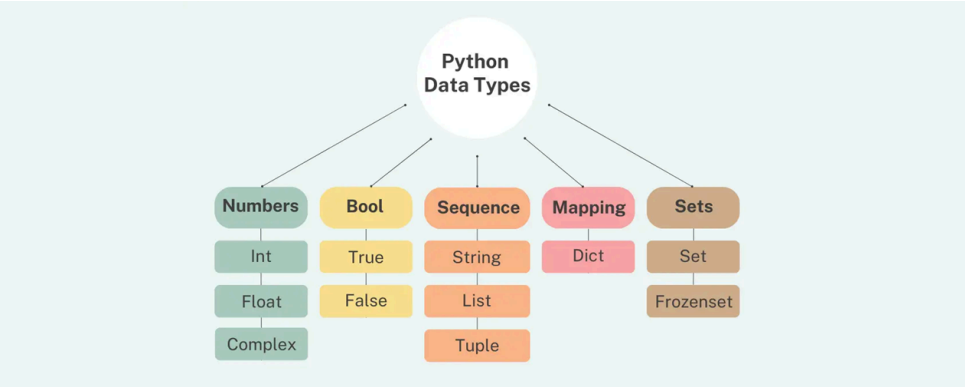
In [ ]:
```

```

Dynamic Types

Key Features of Dynamic Typing:

1. No need to specify type: The type is determined at runtime.
2. Variable type can change: You can assign a different type of value to the same variable.
3. Flexible but requires caution: Because type changes dynamically, errors can occur if not handled properly.

Data Types



Numbers

Integer

In [37]:
```
## Integer
x = 10
y = -5
z = 1000000000

print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'int'>
print(type(z))  # Output: <class 'int'>
```

```
<class 'int'>
<class 'int'>
<class 'int'>
```

In [40]:
```
print(isinstance(x, int))    # Output: True
print(isinstance(y, float))  # Output: True
```

```
True
True
```

Float

In [38]:
```
## Float
a = 3.14
b = -0.5
c = 1.0  # Also considered as a float

print(type(a))  # Output: <class 'float'>
print(type(b))  # Output: <class 'float'>
print(type(c))  # Output: <class 'float'>
```

```
<class 'float'>
<class 'float'>
<class 'float'>
```

Type Conversion

In [4]:
```
x = 10        # int
y = 3.14      # float

# Converting int to float
a = float(x)
print(a, type(a))  # Output: 10.0 <class 'float'>

# Converting float to int
b = int(y)
print(b, type(b))  # Output: 3 <class 'int'>
```

```
10.0 <class 'float'>
3 <class 'int'>
```

In [17]:
```
x = str(100)    # x will be '10'
y = int(100)    # y will be 10
z = float(100)  # z will be 10.0

print( "x =", x )
```

```
print( "y =", y )
print( "z =", z )
```

```
x = 100
y = 100
z = 100.0
```

Bool

In [41]:
```
a = True
b = False

print(type(a))  # Output: <class 'bool'>
print(type(b))  # Output: <class 'bool'>
```

```
<class 'bool'>
<class 'bool'>
```

Python also evaluates other data types (like numbers, strings, lists, etc.) as True or False in a boolean context.

1. 0, None, "" (empty string), [] (empty list), etc. are considered False.
2. All other values (non-zero numbers, non-empty strings, lists, etc.) are considered True.

In [2]:
```
# Numbers
print(bool(0))  # Output: False (0 is considered False)
print(bool(1))  # Output: True  (Non-zero number is considered True)

# Strings
print(bool(""))   # Output: False (Empty string is False)
print(bool("Hello"))  # Output: True (Non-empty string is True)

# Lists
print(bool([]))   # Output: False (Empty list is False)
print(bool([1, 2]))  # Output: True (Non-empty list is True)
```

```
False
True
False
True
False
True
```

Sequence

String

In [30]:
```
# Using single, double, and triple quotes
str1 = 'Hello'
str2 = "World"
```

```
str3 = '''Python is amazing!'''
print(str1, str2, str3)
```

```
Hello World Python is amazing!
```

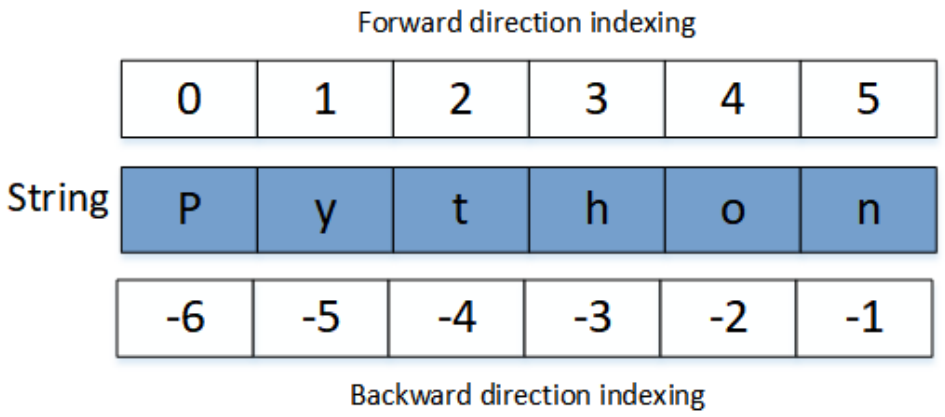
In [32]:
```
print(type(str1))
print(isinstance(str1, str))    # Output: True
```

```
<class 'str'>
True
```

In [21]:
```
multi_line = """Hello World
Python is amazing!"""
print(multi_line)
```

```
Hello World
Python is amazing!
```

1. **Indexing: Access characters using index (0-based).**
2. **Slicing: Extract a substring using [start:end:step].**



In [63]:
```
text = "Python"
print(text[0])    # P (First character)
print(text[5])    # n (Last character)
print(text[-1])   # n (Last character)
```

```
P
n
n
```

In [64]:
```
#print(text[0:6:2])
print(text[-1:-7:-1])
```

```
nohtyP
```

In [71]:
```
print(text[5:None:-1])
print(text[5::-1])
```

```
nohtyP
nohtyP
```

In [29]:
```
# Slicing
print(text[1:4])  # yth (Characters from index 1 to 3)
print(text[:3])   # Pyt (First 3 characters)
print(text[2:])   # thon (From index 2 to end)
print(text[::-1]) # nohtyP (Reverse string)
```

```
yth
Pyt
thon
nohtyP
```

Strings are Immutable in Python

In [36]:
```
text = "Hello"
#text[0] = "M"  #This will raise an error
```

In [37]:
```
# Instead of modifying a string directly, you need to create a new str
text = "Hello"
new_text = "M" + text[1:]  # Creating a new string
print(new_text)  # Output: Mello
```

```
Mello
```

In [61]:
```
# Python provides many built-in methods for string manipulation:
text = "Hello world"
print(text.lower())
print(text.upper())
print(text.title())

text = "Hello world"
print(text.strip()) # Removes whitespace from both sides
print(text.replace('H', 'HH'))

text = "Hello,world"
print(text.split(','))

text = ["Hello","world"]
print(" ".join(text))

text = "Hello world"
print(text.find('o'))
print(text.count('o'))
```

```
hello world
HELLO WORLD
Hello World
Hello world
HHello world
['Hello', 'world']
Hello world
4
2
```

In [2]:
```
text = "Hello world"
print(id(text))
```

```
text1 = text.replace('H', 'HH')
print(id(text1))
```

```
3147826397552
3147825767536
```

In [73]:
```
print(id("Hello"))
```

```
2139240892528
```

In [75]:
```
text = "Hello"
print(id(text))
```

```
2139240892528
```

In [76]:
```
text = text.replace('H', 'HH')  # Assigning new string to text
print(id(text))  # Memory address of the new string
```

```
2139243106608
```

List

A list in Python is a mutable, ordered collection that can store multiple data types, including numbers, strings, and even other lists. Lists are defined using square brackets [].

In [91]:
```
# Empty List
empty_list = []

# List with integers
numbers = [1, 2, 3, 4, 5]

# List with mixed data types
mixed = [10, "Python", 3.14, True]

# List containing another list (Nested List)
nested = [[1, 2], [3, 4], [5, 6]]

# List with duplicate values
duplicates = [1, 2, 2, 3, 4, 4, 5]
```

In [89]:
```
my_list = ["a", "b", "c", "d"] # List
print(my_list[0])  # Output: 'a'
print(my_list[-1]) # Output: 'd' (Last element)
```

```
a
d
```

In [90]:
```
print(my_list[1:3])  # Output: ['b', 'c'] (From index 1 to 2)
print(my_list[::-1]) # Output: ['d', 'c', 'b', 'a'] (Reversed list)
```

```
['b', 'c']
['d', 'c', 'b', 'a']
```

Use a List (list) When:

1. Data needs to be modified (add, remove, or change elements).
2. Ordering matters and may change over time.
3. Operations like sorting, appending, or filtering are needed.
4. Data is expected to grow (tuples are fixed size).

Touple

1. A tuple is a collection of ordered, immutable (unchangeable), and heterogeneous (different data types) elements.
2. Tuples are faster than lists because they are immutable.
3. Tuples are defined using parentheses ().

In [107…
```python
# Empty tuple
empty_tuple = ()

# Tuple with elements
numbers = (1, 2, 3, 4)

# Tuple with different data types
mixed_tuple = (1, "hello", 3.14, True)

# Tuple with one element (comma is required!)
single_element_tuple = (5,)
```

In [108…
```python
# count() - Count occurrences of a value
numbers = (1, 2, 3, 2, 2, 4)
print(numbers.count(2))  # Output: 3
```

```
3
```

In [109…
```python
# index() - Find the index of a value
fruits = ("apple", "banana", "cherry", "banana")
print(fruits.index("banana"))  # Output: 1 (first occurrence)
```

```
1
```

In [110…
```python
# Tuple Slicing
numbers = (0, 1, 2, 3, 4, 5)
print(numbers[1:4])  # Output: (1, 2, 3)
```

```
(1, 2, 3)
```

In [111…
```python
# Tuple Concatenation
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4, 5, 6)
```

```
(1, 2, 3, 4, 5, 6)
```

In [112…
```python
# Tuple Unpacking
person = ("John", 25, "Engineer")
name, age, job = person
```

```
print(name)  # Output: John
print(age)   # Output: 25
print(job)   # Output: Engineer
```

```
John
25
Engineer
```

Use a Tuple (tuple) When:

1. Immutability is needed (data should not change).
2. Performance is important (tuples are faster than lists).
3. Memory optimization is required (tuples use less memory).
4. Tuples represent fixed structures like coordinates, database records, or settings.
5. Tuples can be used as dictionary keys, unlike lists.

Dictionary

A dictionary in Python is an unordered collection of key-value pairs.

1. Each key is unique and is associated with a value.
2. Dictionaries are defined using curly braces {} with the key-value pairs separated by a colon :.
3. Mutable – You can change, add, or remove items.
4. Keys are Unique – No two keys can be the same.

In [115…

```
# Basic Dictionary
student = {
    "name": "John",
    "age": 20,
    "course": "Computer Science"
}
print(student)  # Output: {'name': 'John', 'age': 20, 'course': 'Com
```

```
{'name': 'John', 'age': 20, 'course': 'Computer Science'}
```

In [116…

```
# Dictionary with Different Data Types
person = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "marks": [80, 90, 85]
}
print(person)  # Output: {'name': 'Alice', 'age': 25, 'is_student':
```

```
{'name': 'Alice', 'age': 25, 'is_student': False, 'marks': [80, 90,
85]}
```

Set

A set is an unordered collection of unique elements.

1. Sets are mutable (can be modified), but they do not allow duplicate values.
2. Sets are defined using curly braces {} or the set() constructor.

In [1]:
```
# Basic Set
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output: {'apple', 'banana', 'cherry'}
```

```
{'apple', 'cherry', 'banana'}
```

In [ ]:
```
# Set with Duplicates (Duplicates will be removed)
fruits = {"apple", "banana", "cherry", "apple", "banana"}
print(fruits)  # Output: {'apple', 'banana', 'cherry'}
```

Set Operations

In [12]:
```
# Set Union (|)
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
```

In [13]:
```
# Set Intersection (&)
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1 & set2
print(intersection_set)  # Output: {3}
```

```
{3}
```

In [14]:
```
# Set Difference (-)
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1 - set2
print(difference_set)  # Output: {1, 2}
```

```
{1, 2}
```

In [ ]:
```

```

In [ ]:
```

```

Dynamic Types

Key Features of Dynamic Typing:

1. No need to specify type: The type is determined at runtime.
2. Variable type can change: You can assign a different type of value to the same variable.
3. Flexible but requires caution: Because type changes dynamically, errors can occur if not handled properly.

In [8]:
```
x = 10      # x is an integer
print(type(x))  # Output: <class 'int'>
print(id(x))

x = 3.14    # Now x becomes a float
print(type(x))  # Output: <class 'float'>
print(id(x))

x = "Hello"  # Now x becomes a string
print(type(x))  # Output: <class 'str'>
print(id(x))
```

```
<class 'int'>
140709877826632
<class 'float'>
2139226444464
<class 'str'>
2139227539824
```

Indentation in Python

1. Indentation in Python refers to spaces or tabs at the beginning of a line to define the structure of code blocks.
2. Unlike other languages (C, Java), Python does not use {} (curly brackets) for code blocks. Instead, it relies on indentation.
3. The standard indentation level in Python is 4 spaces per level.

In [87]:
```
age = 18

if age >= 18:
print("You are an adult!")  #IndentationError: expected an indei
```

```
  Cell In[87], line 4
    print("You are an adult!")  #IndentationError: expected an i
ndented block
    ^
IndentationError: expected an indented block after 'if' statemen
t on line 3
```

In [88]:
```
age = 18

if age >= 18:
    print("You are an adult!")  # Correct Indentation
```

```
You are an adult!
```

If Statements

An if-else statement is used to make decisions in Python. It checks a condition:

- If the condition is True, a block of code runs.
- If the condition is False, another block of code runs (if else is used).

In [83]:
```
age = 18

if age >= 18:
    print("You are eligible to vote!")
else:
    print("You are not eligible to vote.")
```

```
You are eligible to vote!
```

In [84]:
```
### if-elif-else (Multiple Conditions)
marks = 85

if marks >= 90:
    print("Grade: A+")
elif marks >= 80:
    print("Grade: A")
elif marks >= 70:
    print("Grade: B")
else:
    print("Grade: C")
```

```
Grade: A
```

In [85]:
```
## Nested if-else (Conditions inside Conditions)

num = 10

if num > 0:
    print("Positive number")
    if num % 2 == 0:
```

```
        print("Even number")
    else:
        print("Odd number")
else:
    print("Negative number or zero")
```

```
Positive number
Even number
```

Comparison/Relational/Conditional Operators

1. == Equal to
2. != Not equal to
3. > Greater than
4. < Less than
5. >= Greater than or equal to
6. <= Less than or equal to

In [77]:
```
x = 10
y = 5

print(x == y)   # False
print(x != y)   # True
print(x > y)    # True
print(x < y)    # False
print(x >= y)   # True
print(x <= y)   # False
```

```
False
True
True
False
True
False
```

Logical Operators

1. **and** Returns True if both conditions are True
2. **or** Returns True if at least one condition is True
3. **not** Reverses the result: True → False, False → True

And

In [79]:
```
x = 7
```

```
print(x > 5 and x < 10)  # True (because 7 is greater than

print(x > 5 and x > 10)  # False (because second condition
```

```
True
False
```

Or

In [80]:
```
x = 3
print(x > 5 or x < 2)   # False OR False → False
print(x > 5 or x == 3)  # False OR True → True
```

```
False
True
```

Not

In [81]:
```
x = 10
print(not(x > 5))  # not(True) → False
print(not(x < 5))  # not(False) → True
```

```
False
True
```

Example

In [ ]:
```
# Real Life Example: 01
age = 20
if age >= 18 and age <= 30:
    print("Eligible for the program")
else:
    print("Not eligible")
```

In [78]:
```
# Real Life Example: 02
marks = 85
if marks >= 80 and marks <= 100:
    print("Grade: A")
```

```
Grade: A
```

In [ ]:
```

```