# Lecture1

September 6, 2024

## 1 Lecture 1: Variables and assignments

#### 1.0.1 Objective:

As soon as we have *data types*, we need *variables* to store the data. In reality, Python does not offer the concept of a variable, but rather that of an *object reference*. As long as the object is immutable (like integers, floats, etc.), there is no significant difference. In this notebook, we discuss the main points surrounding the use of variables in Python.

### 1.1 0. Introduction

In Python, a variable is a fundamental programming construct that facilitates the storage and management of information in the computer's memory. It is assigned a symbolic name, which serves as a reference to the data stored at a particular location in memory. This symbolic name enables you to access and modify the stored information throughout your code with ease.

Variables in Python are defined using the assignment operator =. You can assign a value to a variable directly or assign the value from another variable, making Python flexible in how it handles data. Rather than containing the data themselves, variables in Python act as references to objects in memory. This reference-based approach allows for efficient memory management and provides the flexibility to interact with and modify the data dynamically.

Python's dynamic typing system means that variables can be reassigned to different types of data during program execution. This dynamic nature, combined with the reference model, enhances the language's capability to handle various data types and operations seamlessly. Additionally, Python's garbage collection system manages memory allocation and deallocation, further optimizing performance and resource utilization.

### 1.2 1. Define variable

In Python, there are four fundamental ways to define a variable, each suited to different purposes and contexts:

1. **Direct Assignment**: This is the most straightforward method where a variable is assigned a specific value directly. This method is used for initializing variables with known values that will be used later in the code.
2. **Multiple Assignment**: This technique allows you to assign the same value to multiple variables simultaneously. It is useful for initializing several variables with the same value efficiently.

3. **Parallel Assignment**: This method involves defining multiple variables in a single statement with different values. It is effective for initializing several variables at once with distinct values.

4. **Assignment from Expressions**: Variables can be defined based on expressions involving other variables. This method allows you to create variables that represent calculations or transformations of existing variables.

Each of these methods provides flexibility in how you manage and utilize variables in your code, catering to different scenarios and requirements.

### 1.2.1 1.0. Through direct assignment

```
y = 4.8134  # Defines a variable named y and assigns it the value 4.8134
salutation = "How are you ?"  # Defines a variable named salutation and assigns it the value "
```

This code snippet illustrates the process of creating variables in Python by assigning them specific values. The variable `y` is assigned a floating-point number, while the variable `salutation` is assigned a string value. In Python, this assignment operation creates a reference between the variable name and the data, allowing you to reuse and manipulate the data through the variable name in subsequent parts of your code.

[30]:
```
y = 4.8134  # Defines a variable named y and assigns it the value 4.8134
salutation = "How are you?"  # Defines a variable named salutation and assigns␣
↪it the value "How are you?"
x = 56
```

To display the values of the three defined variables, we use the `print()` function—a function that we will discuss in more detail when we cover function objects in Python. The `print()` function is essential in Python for outputting data to the console, allowing you to see the current state of your variables and debug your code effectively. This function can take multiple arguments, meaning you can print several variables at once, and it automatically converts them to their string representation if necessary.

[31]:
```
print(x)
print(y)
print(salutation)
```

```
56
4.8134
How are you?
```

To display the three values on the same line, you use a single `print()` function, separating the variables with commas. The commas in the print() function automatically insert spaces between the values, making it easy to format your output in a readable manner. This approach allows you to output multiple pieces of data in a single line without needing to manually concatenate strings or add spaces.

[32]:
```
print(x,y,salutation)
```

```
56 4.8134 How are you?
```

### 1.2.2 1.1. Through multiple assignment

The examples presented fall under what we call `direct assignment`. A `multiple assignment` is a specific case of `direct assignment` where the same value is assigned to multiple variables in a single line of code. This technique is useful when you need to initialize several variables with the same starting value efficiently, keeping your code concise and readable.

```
x = y = 7  # x and y are both assigned the value 7 simultaneously.
```

In this line of code, the variables `x` and `y` are assigned the same value of 7 at the same time. This is an example of a `multiple assignment`, where a single value is efficiently assigned to multiple variables in one statement. This technique is particularly useful when initializing variables with the same initial value, ensuring consistency across your code.

[33]:
```
x = y = 7  # x and y are both assigned the value 7 simultaneously.
print(x)
print("======================")
print(y)
```

```
7
======================
7
```

### 1.2.3 1.2. Parallel assignment

A `parallel assignment` involves defining multiple variables using a single equals sign. This technique allows you to assign different values to several variables simultaneously in a single line of code. It enhances code readability and efficiency, especially when you need to initialize multiple variables at once. Example:

```
x, y = 4, 8.33  # Defines two variables, x and y, with values 4 and 8.33 respectively.
```

In this line of code, `x` is assigned the value 4, and `y` is assigned the value 8.33 simultaneously. This is an example of `parallel assignment`, where multiple variables are defined in a single statement. This approach is particularly useful for initializing or updating several variables in a concise and organized manner.

[34]:
```
x, y = 4, 8.33  # Defines two variables, x and y, with values 4 and 8.33␣
↪respectively.
```

### 1.2.4 1.3. Based on other variables

[35]:
```
#Define a variable based on other variables
z1 = x + y  # Defines the variable named z1 and assigns it the sum of variables␣
↪x and y
z2 = x + 5  # Defines the variable named z2 by adding 5 to the value of x
z3 = 2 * y  # Defines the variable named z3 by multiplying the value of y by 2

print(z1,z2,z3)
```

```
12.33 9 16.66
```

[ ]:

Assignment is not comparison! It is important to note that the assignment operator `=` does not have the same meaning as the equality symbol `=` in mathematics. For example, the assignment operator is not symmetric, while the equality symbol is: attempting to swap the order of elements in an assignment statement will inevitably result in an error in the interpreter:

[36]:
```
# Error
128 = a
```

```
  Cell In [36], line 2
    128 = a
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='
```

This brings us to briefly discuss permissible variable names in Python.

**Naming Conventions** Naming conventions for different elements of code are important because they provide additional information to developers about the nature of certain attributes or variables. The conventions for variable names are as follows:

- Reserved keywords such as `if`, `else`, etc., cannot be used as variable names.
- Variable names can start with `_`, `$`, or a letter.
- Variable names can be in lowercase or uppercase.
- Variable names cannot start with a digit.
- White spaces are not allowed in variable names.

A good programmer naturally strives to choose the most meaningful variable names possible.

In Python, there are **33** reserved keywords, and the list is provided below:

| | | | | |
|---|---|---|---|---|
| and | elif | if | or | yield |
| as | else | import | pass | |
| assert | except | in | raise | |
| break | False | is | return | |
| class | finally | lambda | True | |
| continue | for | None | try | |
| def | from | nonlocal | while | |
| del | global | not | with | |

To get the list of reserved keywords in Python, you can use the `keyword` module. Here's how you can do it:

1. **Import the `keyword` module**:

   ```
   import keyword
   ```

2. **Use the `keyword.kwlist` attribute to get the list of keywords**:

```
print(keyword.kwlist)
```

3. **To check if a specific word is a keyword**:

```
print(keyword.iskeyword('if'))  # Returns True
print(keyword.iskeyword('my_var'))  # Returns False
```

The `keyword.kwlist` attribute returns a list of all reserved keywords in Python, and the `keyword.iskeyword()` function checks if a given string is a keyword.

[36]:
```
import keyword
print(keyword.kwlist)
```

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

[37]:
```
print(keyword.iskeyword('if'))  # Returns True
print(keyword.iskeyword('my_var'))  # Returns False
```

```
True
False
```

**Note**: Python is case-sensitive, so variable names `Age` and `age` are considered distinct. Depending on the language, there is a naming convention that is recommended:

- `UpperCamelCase` for class names;
- `CAPITALIZED_WITH_UNDERSCORES` for constants;
- `lowercase_separated_by_underscores` or `snake_case` for other variables.

---

### A Fundamental Exercise: Swapping the Contents of Two Variables

Let's assume that the variables `x` and `y` have the values of integers $\alpha$ and $\beta$ respectively. The goal is to swap the contents of these two variables.

- a. First Method: Propose a method that uses an auxiliary variable `tmp`.

```
tmp = x
x = y
y = tmp
```

- b. Second Method: Execute the following sequence of instructions:

```
x = x + y; y = x - y; x = x - y
```

- c. Third Method (the most "Pythonic"): Use parallel assignment.

```
x, y = y, x
```

---

Example:

[1]:
```
# First Method

x = 5
y = 4
print(x, y)
print("================================================")
print('\t')

tmp = x
x = y
y = tmp
print(x, y)

# Second Method

x = 7
y = 9
print(x, y)
print("================================================")
print('\t')

x = x + y; y = x - y; x = x - y
print(x, y)

# Third Method

x = 90
y = 15
print(x, y)
print("================================================")
print('\t')

x, y = y, x
print(x, y)
```

```
5 4
================================================

4 5
7 9
================================================

9 7
90 15
================================================

15 90
```

It's worth noting that to delete a variable in Python, you can use the `del` function. This function removes the variable from the current namespace, effectively deleting it and freeing up any resources it was using. For example:

```
x = 10  # Define a variable x
del x   # Delete the variable x
```

After executing `del x`, the variable `x` will no longer exist in the current scope, and attempting to access it will result in a `NameError`.

[38]:
```
x = 10  # Define a variable x
print(x)

del x   # Delete the variable x
print(x)
```

```
10
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [38], line 5
      2 print(x)
      4 del x   # Delete the variable x
----> 5 print(x)

NameError: name 'x' is not defined
```

### 1.3 2. Type of a Variable

In Python, the type of a variable refers to the kind of data it holds, such as integers, floating-point numbers, strings, or more complex data structures. Python dynamically assigns the type based on the value assigned to the variable. This type can be determined at any point in the code using the `type()` function.

The type of a variable corresponds to its nature. There are many types of variables (integer, real number, strings, etc.). The most commonly encountered types of variables are integers (`int`), real numbers (`float`), and strings (`str`).

**The basic types include:**

- **None** (nothing)
- **String types:** `str`
  - Enclosed in (single, double, or triple) quotes `'` or `"`: `'Calvin'`, `"Calvin'n'Hobbes"`, `'''Two\nlines'''`, `"""'Why?' he asked."""`
  - Conversion: `str(3.2)`
- **Numeric types:**
  - **Booleans** `bool` (true/false): `True`, `False`
  - **Integers** `int` (no explicit limit value, corresponds to at least C's long type): `-2`, `int(2.1)`, `int("4")`
  - **Reals** `float`

  - **Complex** `complex`: `1+2j`, `5.1j`, `complex(-3.14)`, `complex('j')`
- **Iterable objects:**
  - **Lists** `list`: `['a', 3, [1, 2], 'a']`
  - **Immutable lists** `tuple`: `(2, 3.1, 'a', [])` (depending on the usage, parentheses are not always required)
  - **Keyed lists** `dict`: `{'a':1, 'b':[1, 2], 3:'c'}`
  - **Unordered sets of unique elements** `set`: `{1, 2, 3, 2}`

### 1.3.1 2.0. None (nothing)

The `None` type represents the absence of a value or a null value in Python. It is often used to signify that a variable has no value assigned to it or to indicate the end of a list, function, or loop.

**Example:**

```
x = None
```

[39]:
```
x = None
print(x, type(x))
```

```
None <class 'NoneType'>
```

### 1.3.2 2.1. String Types (str)

Strings in Python are sequences of characters enclosed in quotes. They can be defined using single (`'`), double (`"`), or triple quotes (`'''` or `"""`). Triple quotes allow for multi-line strings.

**Examples:**

```
name = 'Calvin'
quote = "Calvin'n'Hobbes"
multi_line = '''Two
lines'''
```

[40]:
```
name = 'Calvin'
quote = "Calvin'n'Hobbes"
multi_line = '''Two
lines'''

print(name, type(name), '\n')
print('====================================')

print(quote, type(quote),'\n')
print('====================================')


print(multi_line, type(multi_line))
print('====================================')
```

```
Calvin <class 'str'>
```

```
==================================
Calvin'n'Hobbes <class 'str'>

==================================
Two
lines <class 'str'>
==================================
```

### 1.3.3 2.2. Numeric Types

**2.2.0. Booleans (`bool`)** Booleans represent one of two values: `True` or `False`. They are often used in conditional statements to determine the flow of a program.

**Examples:**

```
is_active = True
has_permission = False
```

[41]:
```
is_active = True
has_permission = False


print(is_active, type(is_active), '\n')
print('====================================')

print(has_permission, type(has_permission), '\n')
```

```
True <class 'bool'>

==================================
False <class 'bool'>
```

**2.2.1. Integers (`int`)** Integers are whole numbers without a fractional component. In Python, integers can be of arbitrary precision, meaning they can be as large as the memory allows.

**Examples:**

```
age = 25
negative_number = -42
```

[42]:
```
age = 25
negative_number = -42

print(age, type(age), '\n')
print('====================================')
print(negative_number, type(negative_number))
```

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

```
==================================
-42 <class 'int'>
```

**2.2.2. Reals (`float`)** Floating-point numbers (floats) are numbers with a decimal point. They are used to represent real numbers in Python.

**Examples:**

```
pi = 3.14159
temperature = -2.5
```

[43]:
```
pi = 3.14159
temperature = -2.5


print(pi, type(pi), '\n')
print('==================================')
print(temperature, type(temperature))
```

```
3.14159 <class 'float'>

==================================
-2.5 <class 'float'>
```

**2.2.3. Complex (`complex`)** Complex numbers in Python consist of a real part and an imaginary part. They are represented by `a + bj`, where `a` is the real part and `b` is the imaginary part.

**Examples:**

```
z = 1 + 2j
w = complex(3, -4)
```

[44]:
```
z = 1 + 2j
w = complex(3, -4)

print(z, type(z))
print('==================================')
print(w, type(w))
```

```
(1+2j) <class 'complex'>
==================================
(3-4j) <class 'complex'>
```

### 1.3.4 2.3. Iterable Objects

**2.3.0. Lists (`list`)** A list is an ordered collection of items that can be of different types. Lists are mutable, meaning their contents can be changed after creation.

**Examples:**

```
fruits = ['apple', 'banana', 'cherry']
mixed = [1, 'two', 3.0, [4, 5]]
```

[45]:
```
fruits = ['apple', 'banana', 'cherry']
mixed = [1, 'two', 3.0, [4, 5]]


print(fruits, type(fruits))
print('====================================')
print(mixed, type(mixed))
```

```
['apple', 'banana', 'cherry'] <class 'list'>
====================================
[1, 'two', 3.0, [4, 5]] <class 'list'>
```

**2.3.1. Immutable Lists (tuple)** A tuple is similar to a list but is immutable, meaning its contents cannot be changed after creation. Tuples are often used to store collections of related data.

**Examples:**

```
coordinates = (10.5, 20.8)
colors = ('red', 'green', 'blue')
```

[46]:
```
coordinates = (10.5, 20.8)
colors = ('red', 'green', 'blue')

print(coordinates, type(coordinates))
print('====================================')
print(colors, type(colors))
```

```
(10.5, 20.8) <class 'tuple'>
====================================
('red', 'green', 'blue') <class 'tuple'>
```

**2.3.2. Keyed Lists (dict)** A dictionary is a collection of key-value pairs, where each key is associated with a value. Dictionaries are mutable and allow for fast lookup of values based on their keys.

**Examples:**

```
person = {'name': 'Alice', 'age': 30}
inventory = {'apples': 10, 'bananas': 20}
```

[47]:
```
person = {'name': 'Alice', 'age': 30}
inventory = {'apples': 10, 'bananas': 20}


print(person, type(person))
print('====================================')
print(inventory, type(inventory))
```

```
{'name': 'Alice', 'age': 30} <class 'dict'>
==================================
{'apples': 10, 'bananas': 20} <class 'dict'>
```

**2.3.3. Unordered Sets of Unique Elements (`set`)** A set is an unordered collection of unique elements. Sets are useful for membership tests and eliminating duplicate entries.

**Examples:**

```
unique_numbers = {1, 2, 3, 2}
letters = {'a', 'b', 'c', 'a'}
```

[48]:
```
unique_numbers = {1, 2, 3, 2}
letters = {'a', 'b', 'c', 'a'}

print(unique_numbers, type(unique_numbers))
print('==================================')
print(letters , type(letters ))
```

```
{1, 2, 3} <class 'set'>
==================================
{'c', 'b', 'a'} <class 'set'>
```

### 1.3.5 2.4. Dynamic Typing in Python

Python is a dynamically typed language, meaning that the type of a variable is determined at runtime rather than at compile time. In Python, you don't need to declare the type of a variable when you create it. Instead, the type is inferred based on the value assigned to the variable. This allows for more flexibility but also requires careful handling to avoid type-related errors.

**Key Characteristics of Dynamic Typing:**

- **No Type Declaration**: You simply assign a value to a variable, and Python automatically knows what type it is. `python x = 10 # x is an integer x = "hello" # Now, x is a string`
- **Type Flexibility**: The type of a variable can change over its lifetime. You can reassign a variable to a value of a different type without any issues. `python y = 3.14 # y is initially a float y = True # Now, y is a boolean`
- **Memory Management**: Python handles memory management automatically. When you reassign a variable to a new value, the previous value is discarded if it's no longer referenced elsewhere in the code.

**Pros and Cons of Dynamic Typing:**

- **Pros**:
  - **Flexibility**: You can write more general-purpose code since the type is not fixed.
  - **Ease of Use**: Less boilerplate code, as there is no need for explicit type declarations.
- **Cons**:
  - **Type-Related Errors**: Since types are determined at runtime, it's possible to encounter errors if the wrong type is used in an operation.

- **Performance**: Dynamic typing can be slower than static typing because type checks are done at runtime.

**Example:**

```
# Initially, 'data' is an integer
data = 100

# Now, 'data' is a string
data = "Dynamic Typing"

# And now 'data' is a list
data = [1, 2, 3]

# Python handles these changes without any issues
```

### 1.3.6 2.5. Coercion in Python

**Coercion** in Python refers to the automatic conversion of one data type to another during operations that involve different types. Python is designed to handle these type conversions in a way that makes the language easier to use and reduces the need for manual type casting.

**Key Points About Coercion:**

- **Implicit Coercion**: Python automatically converts one data type to another when necessary to perform an operation. This usually happens in arithmetic operations involving different types, like an integer and a float.
  - For example, if you add an integer to a float, Python will convert the integer to a float before performing the addition.
- **Explicit Coercion**: While Python handles many conversions automatically, you can also manually convert types using built-in functions like `int()`, `float()`, `str()`, etc. This is known as explicit type casting.

**2.5. 0. Implicit Coercion Example:**

```
# Adding an integer and a float
x = 5        # int
y = 3.2      # float

# Python automatically converts 'x' to a float before performing the addition
result = x + y

print(result)  # Output: 8.2 (float)
```

In the example above, Python automatically converts the integer `5` to a float `5.0` to perform the addition with the float `3.2`, resulting in a float `8.2`.

[49]:
```
# Adding an integer and a float
x = 5        # int
y = 3.2      # float
```

```
# Python automatically converts 'x' to a float before performing the addition
result = x + y




print(x, type(x), '\n')
print('====================================')
print(y, type(y),'\n')
print('====================================')
print(result, type(result))  # Output: 8.2 (float)
```

```
5 <class 'int'>

====================================
3.2 <class 'float'>

====================================
8.2 <class 'float'>
```

### 2.5.1. Explicit Coercion Example:

```
# Converting a float to an integer
a = 7.9
b = int(a)  # Explicit coercion using the int() function

print(b)  # Output: 7 (integer, with the decimal part truncated)
```

Here, the float `7.9` is explicitly converted to the integer `7` using the `int()` function, which removes the fractional part.

[50]:
```
a = 7.9
b = int(a)  # Explicit coercion using the int() function

print(b)  # Output: 7 (integer, with the decimal part truncated)
```

```
7
```

### 2.5.2. Common Coercion Scenarios:

- **String to Integer/Float**: When you need to convert a string containing numeric characters to an integer or float.

  ```
  num_str = "123"
  num_int = int(num_str)   # Converts to integer 123
  num_float = float(num_str)  # Converts to float 123.0
  ```

- **Integer/Float to String**: When you need to concatenate a number with a string.

```
age = 25
message = "I am " + str(age) + " years old."
```

- **Boolean to Integer**: `True` is coerced to `1` and `False` to `0` in numeric operations.

```
result = True + 2   # Output: 3 (1 + 2)
```

#### 2.5.3. Pros and Cons of Coercion:

- **Pros**:
  - Simplifies code by reducing the need for explicit type conversions.
  - Makes the language more intuitive and user-friendly.
- **Cons**:
  - Can lead to unexpected results if the automatic type conversion doesn't align with the programmer's intent.
  - Potentially hides bugs related to incorrect data types.

Coercion in Python allows for smoother and more intuitive operations involving different data types. While it adds convenience, it's important to understand how and when Python performs these conversions to avoid unexpected behaviors.

### 1.4 3. Methods associated with variables

In Python, every variable is linked to a variety of attributes and methods that define its behavior and interactions. These methods are functions that are built into the variable's type and allow you to perform various operations on the variable. For example, methods can help you manipulate strings, perform mathematical operations, or interact with lists and dictionaries.

The `dir()` function is useful for exploring these methods and understanding what operations are available for a given variable. By calling `dir()` on a variable, you get a list of all its attributes and methods, including those inherited from its type. This can be particularly helpful for discovering how to use a variable's methods or for debugging.

Here's how you might use `dir()`:

```
# Example with a string variable
text = "Hello, world!"
print(dir(text))

# Example with a list variable
numbers = [1, 2, 3, 4, 5]
print(dir(numbers))
```

In the examples above, `dir(text)` will list methods related to string operations such as `upper()`, `lower()`, and `split()`, while `dir(numbers)` will show methods related to list operations like `append()`, `remove()`, and `sort()`. This feature of Python makes it easier to explore and utilize the functionalities associated with different data types.

[51]:
```
x = 2.5 # Définit une variable numérique x
y = 'my text' # Définit une variable en chaîne de caractères y.
```

Pour afficher l'ensemble des méthodes associées à chacune de ces variables, on fait :

[52]:
```
print(dir(x))
```

```
['__abs__', '__add__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__', '__getattribute__', '__getformat__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__int__', '__le__', '__lt__', '__mod__', '__mul__',
'__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__',
'__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__',
'__rmul__', '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'__trunc__', 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag',
'is_integer', 'real']
```

[53]:
```
print(dir(y))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

You can also type `variable_name.` followed by `TAB` in many interactive Python environments or integrated development environments (IDEs). This action triggers autocompletion, which helps you see and select available methods and attributes associated with that variable. For example:

1. Type `variable_name.` and press `TAB`.
2. A list of methods and attributes that can be used with `variable_name` will appear.

This feature is especially useful for exploring what operations you can perform on a variable and for quickly finding the right method without needing to remember exact method names.

[ ]:
```
# Type x. followed by `TAB`
```

To get help on a specific method in Python, you can use the `help()` function. The syntax is `help(variable_name.method_name)`, where `method_name` is the name of the method you are interested in. For example, if you have a numeric variable of type `float`, which includes a method named `conjugate`, you can obtain information about this method by running:

```
print(help(x.conjugate))
```

This will display documentation about the `conjugate` method, including its purpose and usage, in

the console or terminal.

[54]:
```
print(help(x.conjugate))
```

```
Help on built-in function conjugate:

conjugate() method of builtins.float instance
    Return self, the complex conjugate of any float.

None
```

To display help on all the functions associated with a variable `x`, you simply use:

```
help(x)
```

This command will show the documentation for the type of the variable `x`, including all the methods and attributes available for that type.

[55]:
```
print(help(x))
```

```
Help on float object:

class float(object)
 |  float(x=0, /)
 |
 |  Convert a string or number to a floating point number, if possible.
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __ceil__(self, /)
 |      Return the ceiling as an Integral.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
```

```
|  __floor__(self, /)
|      Return the floor as an Integral.
|
|  __floordiv__(self, value, /)
|      Return self//value.
|
|  __format__(self, format_spec, /)
|      Formats the float according to format_spec.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getnewargs__(self, /)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __int__(self, /)
|      int(self)
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
```

```
 |      Return pow(self, value, mod).
 |
 |  __radd__(self, value, /)
 |      Return value+self.
 |
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __round__(self, ndigits=None, /)
 |      Return the Integral closest to x, rounding half toward even.
 |
 |      When an argument is passed, work like built-in round(x, ndigits).
 |
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |
 |  __rsub__(self, value, /)
 |      Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(self, /)
 |      Return the Integral closest to x between 0 and x.
 |
 |  as_integer_ratio(self, /)
 |      Return integer ratio.
 |
 |      Return a pair of integers, whose ratio is exactly equal to the original
float
```

```
 |      and with a positive denominator.
 |
 |      Raise OverflowError on infinities and a ValueError on NaNs.
 |
 |      >>> (10.0).as_integer_ratio()
 |      (10, 1)
 |      >>> (0.0).as_integer_ratio()
 |      (0, 1)
 |      >>> (-.25).as_integer_ratio()
 |      (-1, 4)
 |
 |  conjugate(self, /)
 |      Return self, the complex conjugate of any float.
 |
 |  hex(self, /)
 |      Return a hexadecimal representation of a floating-point number.
 |
 |      >>> (-0.1).hex()
 |      '-0x1.999999999999ap-4'
 |      >>> 3.14159.hex()
 |      '0x1.921f9f01b866ep+1'
 |
 |  is_integer(self, /)
 |      Return True if the float is an integer.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __getformat__(typestr, /) from builtins.type
 |      You probably don't want to use this function.
 |
 |        typestr
 |          Must be 'double' or 'float'.
 |
 |      It exists mainly to be used in Python's test suite.
 |
 |      This function returns whichever of 'unknown', 'IEEE, big-endian' or
'IEEE,
 |      little-endian' best describes the format of floating point numbers used
by the
 |      C type named by typestr.
 |
 |  fromhex(string, /) from builtins.type
 |      Create a floating-point number from a hexadecimal string.
 |
 |      >>> float.fromhex('0x1.ffffp10')
 |      2047.984375
 |      >>> float.fromhex('-0x1p-1074')
```

```
|      -5e-324
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  imag
|      the imaginary part of a complex number
|
|  real
|      the real part of a complex number

None
```

### 1.4.1 4. Arithmetic and Logical (Boolean) Operators

In Python, two major categories of operators are used to define variables and instructions: arithmetic operators and logical (boolean) operators.

- **Arithmetic Operators**: These operators perform common mathematical operations. They include addition, subtraction, multiplication, division, and others that are essential for numerical computations.
- **Logical (Boolean) Operators**: These operators are used for comparing values and evaluating logical expressions. A boolean value represents one of two possibilities: true or false. Boolean values result from evaluating logical expressions and are used to make decisions within a program, such as executing certain actions when specific conditions are met.

**Boolean values** are crucial for control flow in programming, allowing for conditional execution based on whether a condition evaluates to true or false.

#### 4.0. Arithmetic Operators

| Operation | Symbol | Example |
|---|---|---|
| Addition | `+` | `x = 2 + 3` |
| Subtraction | `-` | `z = x - y` |
| Multiplication | `*` | `y = 3 * x` |
| Real Division | `/` | `5 / 2 = 2.5` |
| Integer Division | `//` | `5 // 2 = 2` |
| Exponentiation | `**` | `x ** 2 = x * x` |
| Modulo (Remainder) | `%` | `17 % 3 = 2` |
| Increment Addition | `+=` | `x += 4` (i.e., `x = x + 4`) |
| Increment Subtraction | `-=` | `x -= 4` (i.e., `x = x - 4`) |

### 4.1. Logical Operators<

| Operation | Symbol | Description | Example |
|---|---|---|---|
| Logical AND | `and` | Returns `True` if both operands are true | `True and False` yields `False` |
| Logical OR | `or` | Returns `True` if at least one operand is true | `True or False` yields `True` |
| Logical NOT | `not` | Returns `True` if the operand is false | `not True` yields `False` |
| Logical XOR (Exclusive OR) | `^` | Returns `True` if operands are different | `True ^ False` yields `True` |
| Logical equality | `==` | Returns `True` if both operands are equal | `x == y` |
| Logical inequality | `!=` | Returns `True` if operands are not equal | `x != y` |
| Less than | `<` | Returns `True` if left operand is less than right operand | `x < y` |
| Greater than | `>` | Returns `True` if left operand is greater than right operand | `x > y` |
| Less than or equal to | `<=` | Returns `True` if left operand is less than or equal to right operand | `x <= y` |
| Greater than or equal to | `>=` | Returns `True` if left operand is greater than or equal to right operand | `x >= y` |

To access the complete list of standard Python operators and their equivalent functions, see this page. You can also refer to this page for some examples of standard operator usage.

#### 1.4.2 5. User input ( the `input()` function)

In Python, the `input()` function is used to capture user input from the console. It pauses the program's execution and waits for the user to type something, which is then returned as a string. This input can be stored in a variable, allowing you to use the entered data later in your code.

**5.0. Basic Usage** The basic syntax for the `input()` function is:

```
variable_name = input(prompt)
```

- **`prompt`**: This is an optional argument. It is a string that is displayed to the user, providing instructions or asking for specific input.
- **`variable_name`**: This is the variable that will store the value entered by the user.

**Example** Here's a simple example of using the `input()` function:

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

In this example: - The program prompts the user to enter their name. - The entered name is stored in the variable `name`. - The program then greets the user using the name provided.

[1]:
```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

```
Enter your name: Max
Hello, Max!
```

### 5.1. Important Notes

- **Type Conversion**: Since `input()` always returns the input as a string, you may need to convert it to the appropriate type (e.g., `int`, `float`) depending on the context.

  ```
  age = int(input("Enter your age: "))
  ```

- **Handling Errors**: When converting input, it's important to handle potential errors, such as the user entering a non-numeric value when an integer is expected.
- **Security Considerations**: Be cautious when using `input()` in sensitive applications, as it can introduce security risks if the input is not properly validated or sanitized.

Using the `input()` function is a common way to make your Python programs interactive, enabling users to provide data that can be processed and utilized by the program.

**5.2. Accepting User Inputs (as both integer and string)** **`input(prompt)`** prompts for and returns input as a string. Hence, if the user inputs a integer, the code should convert the string to an integer and then proceed.

[4]:
```
a = input("Hello, \nHow are you?  ") # \n means new line


print("================================== \n")
print(type(a))
```

```
Hello,
How are you?  Fine
==================================

<class 'str'>
```

[5]:
```
try_something = input("Type something here and it will be stored in variable␣
 ↪try_something \t")

print("================================== \n")
print(type(try_something))
```

```
Type something here and it will be stored in variable try_something     Gababa
==================================
```

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

[ ]:
```
number = input("Enter number: ")
name = input("Enter name: ")

print("\n")
print("Printing type of a input value")
print("type of number", type(number))
print("type of name", type(name))
```

**5.3. `eval()` (accepting user inputs; only as integer)** The `eval()` function in Python can be used in conjunction with `input()` to evaluate a string as a Python expression. This can be particularly useful when you want to allow the user to input a mathematical expression or Python code directly and have it evaluated at runtime.

**Basic Usage of `eval()`** The basic syntax for using `eval()` with `input()` is:

```
result = eval(input(prompt))
```

- **`prompt`**: This is the text displayed to the user to guide them on what to input.
- **`result`**: This variable stores the output after evaluating the user input as a Python expression.

**Example** Here's an example where the user is allowed to input a mathematical expression, and `eval()` evaluates it:

```
expression = input("Enter a mathematical expression: ")
result = eval(expression)
print("The result is:", result)
```

[11]:
```
expression = input("Enter a mathematical expression: ")
result = eval(expression)
print("The result is:", result)
```

```
Enter a mathematical expression: 63.8
The result is: 63.8
```

In this example: - The program prompts the user to enter a mathematical expression. - The input is passed to `eval()`, which evaluates the expression. - The result is then printed out.

**Important Notes**

- **Use with Caution**: The `eval()` function can be dangerous if used with untrusted input, as it will execute any code passed to it. This could potentially lead to security vulnerabilities, such as code injection attacks. It should only be used in safe, controlled environments where the input is trusted.
- **Valid Python Expressions**: The string passed to `eval()` must be a valid Python expression. If the string contains syntax errors or invalid operations, Python will raise an exception.

- **Alternative**: In many cases, using `int()` or `float()` for type conversion, or safely parsing and evaluating input without `eval()`, may be preferable for security reasons.

Using `eval()` with `input()` can be powerful for dynamic code evaluation, but it should be used responsibly to avoid unintended consequences.

### 1.5 Further reading

0. Programming with Python; Section 4
1. Programming with Python; Section 5

< 0. Introduction | ToC | 2. Strings >