

Variables and Simple Data Types

Variables in Python are names you give to values to store them in memory. These values can be of different data types. The basic types of data include:

1. Numbers

- **Integers:** Integers are characterized by the absence of fractions or decimal numbers and include positive numbers, negative numbers, and zero (e.g., 1, -5, 100, 0).
 - **Floating-Point Numbers:** used to store numbers with decimal points (e.g., 3.14, -0.5).
2. **Strings:** used to store sequences of characters (e.g., "Hello, World!").
 3. **Booleans:** used to store only two values: **True** or **False**.

1. Numbers

Numbers are often used in programming to record scores in games, represent data in visualization, store information in web applications, etc. Python treats numbers in several different ways, depending on how they are used. Let's first look at how Python manages integers, because they are the easiest to work with.

```
x = 5 # input
y = -10 # input
z = 123456789 # input

print(x) # Output: 5
print(y) # Output: -10
print(z) # Output: 123456789
```

- **Integers**

You can add (+), **subtract** (-), **multiply** (*) and, **divide** (/) integers in Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Python uses two multiplication symbols (******) to represent **exponents**:

```
>>> 3 ** 2 # (32)
```

```
9
```

```
>>> 3 ** 3 # (33)
```

```
27
```

```
>>> 10 ** 6 # (106)
```

```
1000000
```

Priority: Python supports the **order of operations** too, so you can use **multiple operations** in one expression. You can also use **parentheses “()”** to modify the order of operations so Python can evaluate your expression in the order you specify. **For example:**

```
>>> 2 + 3*4
```

```
14
```

```
>>> (2 + 3) * 4
```

```
20
```

```
>>> 5-10+ (4-8)/4**2*(3+2) # -5+[-4/16]*5 == -5+[-0.25]*5 == -5-1.25 = -6.25
-6.25
```

Priority

- 1- parentheses
- 2- exponents
- 3- divide(/)
- 4- multiply (*)
- 5- add (+) and subtract (-)

• Floating-Point Numbers

Python calls any number with a decimal point a **float**. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number,

Example: 3.14159 **Or** 0.1 **Or** 100.0

Arithmetic operations using decimal numbers

```
>>> 0.1 + 0.1
0.2
```

```
>>> 0.2 + 0.2
0.4
```

```
>>> 2 * 0.1
0.2
```

```
>>> 2 * 0.2
0.4
```

```
>>> 0.5 / 0.2
2.5 # 0.2 ÷ 0.5 = 2.5
```

```
>>> 1.0 / 3.0
0.3333333333333333
round(1.0 / 3.0)
```

```
0.3
>>> 0.5 ** 2 # exponents
```

```
0.25
```

بايثون يخفي هذا الخطأ الصغير في العرض في بعض الاحيان لتسهيل القراءة، لذلك لا تراه عادة كما في $0.1 + 0.1$.

✦ أما في حالات أخرى (مثل جمع $0.2 + 0.1$ أو عمليات أكثر تعقيداً)، فإن الخطأ يصبح أوضح ويظهر عند الطباعة مثل: $3.0 / 1.0$

في بعض الأحيان قد تظهر نتائج غير متوقعة اي **اعداد اضافية** بسبب طريقة تمثيل الأعداد داخل الكمبيوتر. على سبيل المثال:

عند جمع 0.2 مع 0.1 قد تحصل على نتيجة مثل:
 0.30000000000000004 بدلاً من 0.3 بالضبط. كذلك عند ضرب 3 في 0.1 قد تحصل على نفس النتيجة. وهذا لا يظهر مع جميع الأرقام انما عند بعض الأرقام، حيث لا يظهر عند $1.0/2.0$ لكنه يظهر عند $1.0/3.0$ اي انه يظهر الباقي.

هذا النوع من النتائج يحدث في جميع لغات البرمجة وليس بايثون فقط. السبب هو أن الكمبيوتر يمثل الأعداد باستخدام النظام الثنائي (binary system)، وهو غير مثالي لتمثيل بعض الأعداد العشرية بدقة تامة. يمكن حل هذه المشكلة و التحكم بالدقة تقريبا المنازل العشرية) عن طريق الابعاز round .

الرقم 4 في النهاية ناتج عن هذا الخطأ الصغير جداً في التقريب (يسمى floating-point rounding error) — أي أن الناتج الحقيقي داخل الكمبيوتر أكبر قليلاً من 0.3.

2. Strings

```
greeting = "Hello, World!" # Input  
name = 'Alice' # Input
```

```
print(greeting) # Output: Hello, World!  
print(name) # Output: Alice
```

Combining or Concatenating Strings

You can concatenate strings using the plus sign (+).

Example:

```
first_name = "ada"  
last_name = "lovelace"  
full_name = first_name + " " + last_name  
print("Hello," + full_name.title() + "!")
```

العلامة " تُستخدم لتحديد بداية ونهاية السلسلة النصية في بايثون، أي لتعريف أن المحتوى بينهما هو نص (string). و إذا لم يتم كتابة العلامة (") يظهر خطأ لأن بايثون لا يعرف ما هو ada كمتغير.

فائدة علامتي الاقتباس " " في هذا السطر هي إضافة فراغ بين الاسم الأول والاسم الأخير عند دمجهما . و يجب ان تكون هناك مسافة بين علامتي الاقتباس.

Result: "Hello, Ada Lovelace!" # Hello : optional additional text, title(): to capitalize the first letter

3. Booleans

```
is_student = True  
is_teacher = False
```

```
print(is_student) # Output: True  
print(is_teacher) # Output: False
```

Additional notes:

The `type()` function can be used to determine the data type of any variable

```
a = 5  
b = 3.14  
c = "Hello"  
d = True
```

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

Variable Naming Rules in Python

When naming variables in Python, there are some rules and guidelines must be follow to ensure that the names are valid and understandable to readers. These rules help in writing clean, readable, and maintainable code.

- Variable names can only contain letters, numbers, or underscores (_).
- Names must not start with a number.
- Reserved words in Python, such as `print`, `if`, `else`, `etc.`, should be avoided.
- Names should be clear and descriptive.

تسمية جيدة : total_score
تسمية سيئة: ts

Examples:

➤ Valid Variable Names

```
user_age = 25          # snake_case: لفصل بين الكلمات
userName = "totalPrice" # camelCase: تبدأ الكلمة الأولى بحرف صغير
total_price = 99.99
is_student = True
_country = "Egypt"
```

➤ Invalid Variable Names

- class (uses a reserved keyword) # `class = "Alaa"` # خطأ، لا يمكن استخدام الكلمة المحجوزة
- 1st_variable (cannot start with a number) # `1st_variable = 10` # خطأ
- total amount (contains a space) # `total amount = 100` # خطأ اسم المتغير لا يمكن أن يحتوي على مسافة
- my-variable (contains a hyphen) # `my-variable` # لا يمكن أن يحتوي على شرطة (-) في بايثون، لأنها تُستخدم كعامل الطرح
- if (uses a reserved keyword) # لا يمكن استخدام الكلمة المحجوزة
- age# (contains an invalid character) # `age#` # المتغير لا يمكن أن يحتوي على أحرف غير صالحة مثل #
- user@name (contains an invalid character) # `user@name` # يجب ان لا يحتوي على @
- None (uses a reserved keyword) # `None` # كلمة محجوزة في بايثون
- total\$amount (contains an invalid character) # `total$amount` # \$
- my variable (contains a space)
- -score (cannot start with a hyphen)

Some Additional Guidelines:

Use `_` to avoid conflicts with reserved keywords: If the desired variable name conflicts with a reserved keyword, you can use an underscore (`_`) at the beginning of the name.

Example: `_class` instead of `class`.

Avoiding Name Errors When Using Variables

Name errors occur when trying to use a variable that has not been defined previously or if there is a typo in the name.

Example:

- **Using an Undeclared Variable:** Attempting to use a variable before assigning it a value.

```
print(my_variable) # NameError: name 'my_variable' is not defined
```

- **Typographical Errors:** Mistakes in spelling the variable name.

```
myVariable = 10
```

```
print(myVariabel) # NameError: name 'myVariabel' is not defined
```

- **Case Sensitivity:** Variable names in Python are case-sensitive, so `myVariable` and `myvariable` are treated as different variables.

```
myVariable = 10
```

```
print(myvariable) # NameError: name 'myvariable' is not defined
```

To avoid name errors, ensure that variables are defined before use and that their names are spelled correctly and consistently.

Changing Case in a String with Methods

Python provides built-in functions to change the case of letters in a string, such as:

- **`title()`:** Capitalizes the first letter of each word.
- **`upper()`:** Converts all letters to uppercase.
- **`lower()`:** Converts all letters to lowercase.

Example:

```
name = "ada lovelace"
print(name.title()) # Result: Ada Lovelace
print(name.upper()) # Result: ADA LOVELACE
print(name.lower()) # Result: ada lovelace
```

Adding Whitespace to Strings with Tabs or Newlines

Text can be organized and formatted more effectively when printing strings by using special characters such as `\n` for a new line and `\n\t` to add a white space (Tab).

Examples:**1. Adding a new line using `\n`**

```
print("Languages:\nPython\nC\nJavaScript")
```

Result:

```
Languages:
Python
C
JavaScript
```

2. Adding Whitespace using `\n\t`

```
print("Languages:\n\tPython\n\tC\n\tJavaScript")
```

Result:

```
Languages:
    Python
    C
    JavaScript
```

تعمل `\n` سطر جديد اما `\n\t` فتعمل سطر جديد مع مسافة بادئة
اي الفرق بين الاثنين هو الحرف `t` الذي يعني Tab اي بادئة.

3. Combine the two cases together

```
print("Favorite Languages:\n\tPython\n\tJava\n\tJavaScript\nGo\nRust")
```

Result:

```
Favorite Languages:
    Python
```

Java
JavaScript
Go
Rust

Stripping Whitespace

To remove unnecessary white spaces from text, you can use the following:

- `rstrip()`: Removes spaces from the end of the text.
- `lstrip()`: Removes spaces from the beginning of the text.
- `strip()`: Removes spaces from both ends.

Example:

```
favorite_language = ' python '  
print(favorite_language.rstrip()) # 'python'  
print(favorite_language.lstrip()) # 'python '  
print(favorite_language.strip()) # 'python'
```

Avoiding Type Errors with the str() Function

If you attempt to combine numbers with strings directly, it will result in an error. You must convert numbers to strings using the `str()` function. In the example below, the number 23 (an integer) is converted to a string using `str()` so it can be combined with the text string.

Note: Python cannot convert strings that contain both letters and numbers into integers. Instead, you should convert numbers into strings, not the other way around.

Examples:

```
age = 23  
message = "Happy " + age + "rd Birthday!"  
print(message) # Result = error  
  
message = "Happy " + str(age) + "rd Birthday!"  
print(message) # يطبع "Happy 23rd Birthday!"
```

Implicit conversion: Some languages perform it automatically between different types (strings and integers).

In Python: Explicit conversion must be used with functions like `str()` to convert integers to strings.

Python understands numbers as integers or decimals by default, but if they are written inside quotation marks ('10'), it treats them as strings. They must be converted to integers using `int()` when needed to be used as numbers.

```
num1 = "10"
num2 = "10"
sum =(num1) + (num2)
print(sum)# Result= 1010
sum = int(num1) + int(num2) # convert string to number
print(sum)# Result= 20
```

If the quotation marks are removed, the numbers will be considered as integers or decimals by default in Python, and there is no need to convert them using `int()`.

```
num1 = 10
num2 = 10
result = num1 + num2
print(result) # Result= 20
```

If the string contains both string and numbers, attempting to convert it to a number using `int()` will result in an error, but it can be handled as a literal string.

```
num_str = "10abc"

print(num_str) # There is no error here, the string will be printed as is

Note: "String values must be enclosed in quotation marks when defined in Python."

num_str = "10abc"
num = int(num_str) # This will result in an error because it contains string
```

Comments

Comments help explain and document the code, and they are not executed by the interpreter. Comments are written using the `#` sign.

Example:

```
print("Hello Python people!") # It prints a welcome message
```