In Python, dynamic typing is a feature where the type of a variable is determined at runtime, rather than at compile-time.

This means that the type of a variable can change throughout the execution of a program. For example, you can store an integer in a variable, and then later store a string in the same variable. This is different from statically typed languages, where the type of a variable is determined at compile-time and cannot be changed throughout the execution of the program. It is important to note that while Python is dynamically typed, it is also strongly typed, meaning that variables cannot change type accidentally. However, it is still possible to explicitly convert variables from one type to another using type casting.

Important for integer division //

Always ignore the decimals so

print(11 // 5) # 2

but print(-11 // 5) # -3

**Also for the negative**

# With negative numbers, it preserves the divisor sign

print(-11 % 5) # 4

print(11 % -5) # -4

(x % y) == x - (x // y) \* y

11 % -5 == 11 - (-3) \* (-5) == 11 - 15 == -4

**Operation priority**

To sum up, there is a list of priorities for all considered operations:

1. power
2. unary minus
3. multiplication, division, and remainder
4. addition and subtraction

**print(10 / 5 / 2) # 1.0**

**print(8 / 2 \* 5) # 20.0**

Python follows a left-to-right operation convention from mathematics. It's a good thing to know, so try to keep that in mind, too!

**PEP time!**

There are a few things to mention about the use of [binary](https://hyperskill.org/learn/step/5865) operators (that is, the operators that influence both [operands](https://hyperskill.org/learn/step/5865)). As you know, readability does matter in Python. So first, remember to surround a binary operator with a single space on both sides:

number=30+12 # No!

number = 30 + 12 # It's better this way

**Operators** are special symbols indicating what operation to perform. **Operands**are values that the operation is performed on. Let's consider our example: 30 + 12. Here + is an operator and 30 and 12 are operands.

Also, sometimes people use the break **after**binary operators. But this can hurt readability in two ways:

* the operators are not in one column,
* each operator has moved away from its operand and onto the previous line:

# No: operators sit far away from their operands

income = (gross\_wages +

taxable\_interest +

(dividends - qualified\_dividends) -

ira\_deduction -

student\_loan\_interest)

Mathematicians and their publishers follow the opposite convention in order to solve the readability problem. Donald Knuth explains this in his Computers and Typesetting series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations". Following this tradition makes the code more readable:

# Yes: easy to match operators with operands

income = (gross\_wages

+ taxable\_interest

+ (dividends - qualified\_dividends)

- ira\_deduction

- student\_loan\_interest)

a = int(input())

b = int(input())

print(a+b)

**Square root** **Does NOT have its own operator!** Instead, you need to use the math.sqrt() function or exponentiation (x \*\* 0.5).

Example:

python

CopyEdit

import math

print(math.sqrt(16)) # Output: 4.0

or

python

CopyEdit

print(16 \*\* 0.5) # Output: 4.0

user\_name = input()

print('Hello, ' + user\_name)

 **Variables & Functions** → snake\_case (all lowercase with underscores)  
Example: my\_variable, calculate\_sum()

 **Classes** → PascalCase (also called UpperCamelCase)  
Example: MyClass, DataProcessor

 **Constants** → ALL\_CAPS (all uppercase with underscores)  
Example: MAX\_SPEED, PI

 **Module & Package Names** → snake\_case (lowercase, no spaces)  
Example: my\_module.py, data\_processor.py

 **Private Variables/Functions** → Prefix with underscore (\_)  
Example: \_internal\_function(), \_hidden\_variable

 **Dunder (Double Underscore) Methods** → Used for special functions  
Example: \_\_init\_\_(), \_\_str\_\_()

What else should you remember? Well, this: any value you enter, the function sees as a **string**. It doesn't matter if you enter digits or letters, the input will be converted to a string.

If you want an input to be a **number**, you should write it explicitly:

print("What's your favorite number?")

value = int(input()) # now value keeps an integer number

However, be careful: in these circumstances, if a user enters a non-integer value, an Error will appear.

To read several inputs, you should call the function more than once:

day = int(input()) # 4

month = input() # October

Remember that input() gives you a str, not an int

 As a general rule, they are explicitly converted to corresponding [numerical types](https://hyperskill.org/learn/step/5872):

integer = int(input())

floating\_point = float(input())

# simple assignment

number = 10

number = number + 1 # 11

# compound assignment

number = 10

number += 1 # 11

Naturally, similar assignment forms exist for the rest of arithmetic operations: -=, \*=, /=, //=, %=, \*\*=. Given the opportunity, use them to save time and effort.