

02 – Basic scalar data types

Bálint Aradi

Scientific Programming in Python (2025)

<https://atticlectures.net/scipro/python-2025/>

Immutable data types

- Can not be changed once they have been created
- You must create a new (changed) instance if you want to change them
- Examples: bool (True, False), integer, float, string, tuple, frozen set, etc.

Mutable data types

- Their content can be changed after their creation
- Examples: list, set, dictionary, file, etc.
- Handling of mutable data types can have certain “**side-effects**”

Integers (int)

- Range: arbitrary
- If value is beyond the **long int** data type in C (2^{63} on 64 bit machines), operations become rather slow (runs emulated, not natively)

```
%%timeit -r 10  
num = 2**3625  
for ii in range(63):  
    num *= 2
```

Runs the cell a given amount of time and measures execution time

Compare

```
%%timeit -r 10  
num = 2**0  
for ii in range(63):  
    num *= 2
```

% Jupyter kernel "magic" commands

Floating point numbers (float, complex)

Real numbers

- The same as **double type in C**
 - Range: +/-1E-323 – +/-1E+308
 - Precision: 16 digits
- Can be entered either in **fixed** or in **exponential** notation

```
>>> 0.123
0.123
>>> 1.23E-1
0.123
>>> 9e-1300
0
>>> 9e1000
inf
```

Complex numbers

- Represented by a **pair of real numbers**
- Real and imaginary part have the same range then usual real numbers
- Input as ***RealPart + ImaginaryPartJ***

```
>>> 2.0 + 3.3j
(2+3.3j)
```

Arithmetic operators

<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>//</code>	Integer division
<code>%</code>	Division remainder
<code>-</code>	Negation
<code>**</code>	Power

```
>>> 1 + 2
3
>>> 3 - 4
-1
>>> 5 * 6
30
>>> 5 / 2
2.5
```

```
>>> 5 // 2
2
>>> 5 % 2
1
>>> -8
-8
>>> 2**0.5
1.4142135623730951
```

Relation operators

<code>==</code>	equal
<code>!=</code>	unequal
<code><</code>	less
<code><=</code>	less equal
<code>></code>	greater
<code>>=</code>	greater equal

Comparison gives bool type
as result (True/False)



```
>>> 3 == 2
False
>>> 3 != 2
True
>>> 3 < 2
False
>>> 3 > 2
True
>>> 3 >= 2
True
>>> 3 <= 2
False
```

```
>>> 3.0+2j == 3.0+3j
False
>>> 3.0+2j != 3.0+3j
True
>>> 3.0+2j < 2.0-1.2j
Traceback (most recent call...
```



Error: Complex numbers can not be ordered

Comparing with `==` or `!=` is OK

Booleans (bool) & logical operators

- They are actually numbers, only shown differently
 - **False**: 0, **True**: 1

```
>>> True
True
>>> False
False
>>> 2 * True
2
```

Logical operators

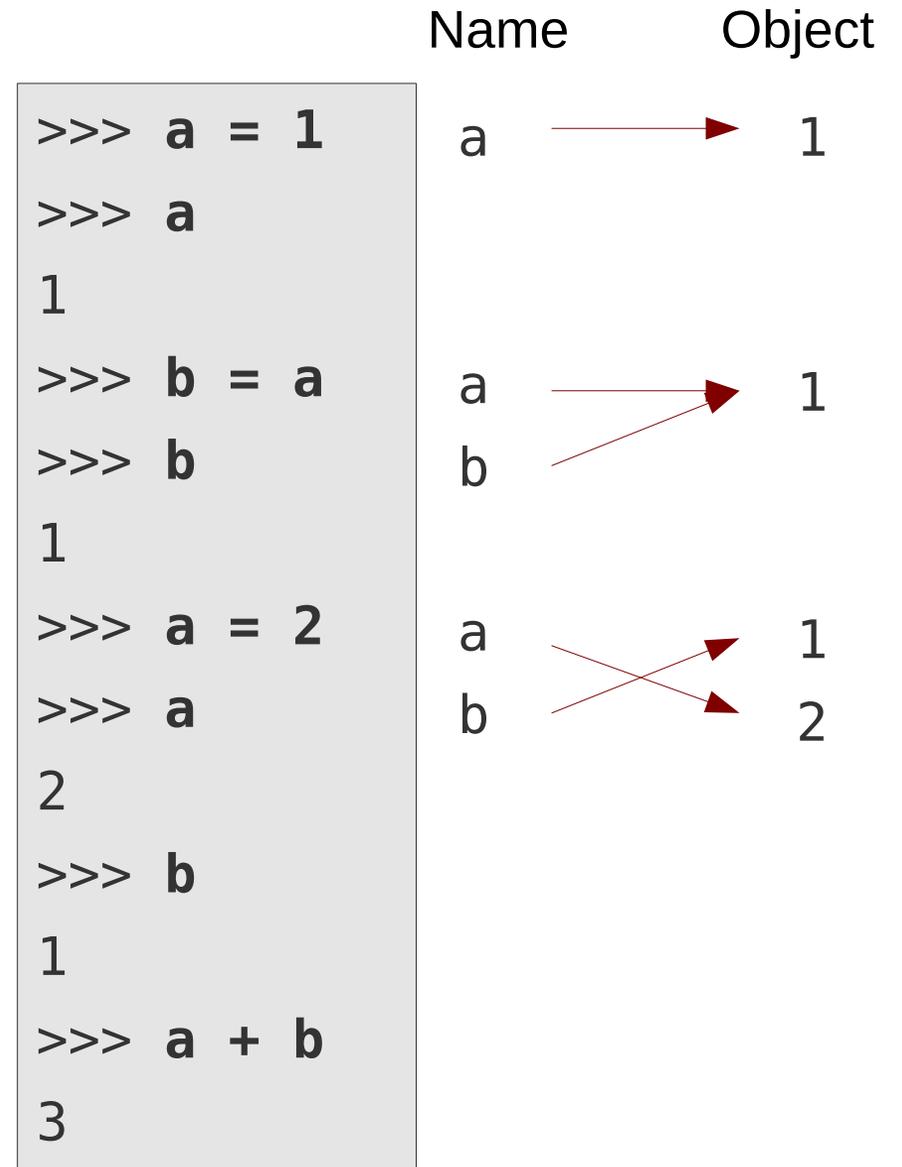
- Logical **AND** (True if both operands True)
- Logical **OR** (True if any of the operands True)
- Logical **NOT** (Negates operand)

```
>>> True and False
False
>>> False or True
True
>>> not True
False
```

- In Python each object can serve as a logical value (details later)

Assignment

- An object (e.g. result of an operation) gets a **name assigned** (variable name)
- **Name = Object**
Name points to / aliases Object
- **Name1 = Name2**
Name1 points to the same object which Name2 points to
- When using a variable name in an expression, it will be substituted with the object it points to.
- There are **no “classic” variables** in Python, just **pointers / aliases!**



Strings

- Strings are specified between **apostrophes or quotes**:

```
>>> name1 = 'john'  
>>> name2 = "tom"  
>>> name1  
'john'  
>>> name2  
'tom'
```

- **Length of a string** can be queried by the **len()** function:

```
>>> len(name1)  
4
```

- Multiline strings can be specified between **triple apostrophes or quotes**:

```
>>> longstr = """First line  
... followed by the second"""  
>>> longstr  
'First line\nfollowed by the second'
```

newline character

Strings

- **Parts of a string** can be accessed by the `[]` operator:

Elements are enumerated **starting with zero**

When selecting ranges as `[lower:upper]`, the **lower bound is inclusive** the **upper bound is exclusive**

Range increment can be also specified with `[lower:upper:increment]`

When lower bound is omitted, range starts from the very first element (0 – range increment pos., last – range increment neg.)

When upper bound is omitted, range ends beyond last element (last element is included)

Negative range increment: iterating backwards

Empty range returns empty string

```
>>> txt = "some text"
>>> txt[0]
's'
>>> txt[0:4]
'some'
>>> txt[0:9:2]
'sm et'
>>> txt[:4]
'some'
>>> txt[4:]
'text'
>>> txt[8:4:-1]
'txet'
>>> txt[3:3]
''
```

Strings

- Strings are **immutable**, they can not be changed once created:

```
>>> txt[0] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str'...does not support item assignment
```

- Strings can be **concatenated** by the **+** operator or by whitespace for string literals:

```
>>> name1 + " " + name2
'john tom'
>>> "str1" "str2"
'str1str2'
```

- Strings can be **repeated** by the ***** operator:

```
>>> "ab" * 3
'ababab'
```

String formatting, f-strings

- Formatted strings (**f-strings**): String containing expressions with optional formatting options
- Expressions are enclosed in {}

```
aa = 12
bb = 135
print(f"a = {aa}, b = {bb}")
```

a = 12, b = 135

- Optional formatting options can be specified after the expression, separated by a colon (:)

```
print(f"a = {aa:3d}\nb = {bb:3d}")
```

Newline character

Field width

Data type

a = 12

b = 135

- Data type must match expression type:

```
cc = 12.35
print(f"c = {cc:4d}")
```

ValueError: Unknown format code 'd'
for object of type 'float'

Few formatting options

<code>:Wd</code>	integer number
<code>:W.Pf</code>	floating point number in fixed notation
<code>:W.Pe</code>	floating point number in exponential notation (with small e)
<code>:W.PE</code>	floating point number in exponential notation (with capital E)
<code>:W.Pg</code>	:f or :e depending on the value of the floating point
<code>:W.PG</code>	:f or :e depending on the value of the floating point
<code>:Ws</code>	string (converts given object to a string)

W (width) minimal field width (optional)

P (precision) number of decimal places (optional)

```
ff = 1.2
```

```
f"{ff:12.4E}"
```

```
f"{ff:12E}"
```

```
f"{ff:.4E}"
```

```
ss = "ab"
```

```
f"{ss:5s}"
```

```
' 1.2000E+00 '
```

```
'1.200000E+00 '
```

```
'1.2000E+00 '
```

Numbers
aligned
right

```
' ab '
```

String aligned left

For further formatting options see the [format specification mini-language](#)

Few remarks on string formatting

- If the field width is too small for the given representation, it will be automatically expanded

```
num = 123  
f" |{num:1d} | " → ' |123| '
```

- If you need literal curly braces in the formatted string, they must be doubled:

```
num = 123  
f" {{{{0:d}}}" → '{123}'
```

- Formatted strings can be created with the **.format()** method as well
- Expressions are given as parameters of the **.format()** method

```
num = 123  
" |{:4d} | ".format(num) → ' | 123| '
```

- Parameters can be referred by their position

```
num1 = 12  
num2 = 34  
" |{0:d}, {1:d}, {0:d} | ".format(num1, num2)
```

' |12, 34, 12| '

Converting data types into each other

- Each data type has a special function, which tries to convert its argument into an object with the given data type:

`int()`, `float()`, `complex()`, `str()`

- Argument can have arbitrary data type
- If the conversion fails, an exception is raised (error)

```
>>> int(3.2)
3
>>> float("12.1")
12.1
>>> complex("3+2j")
(3+2j)
>>> complex("3.0+2.0j")
(3+2j)
```

```
>>> valstr = "3"
>>> int(valstr)
3
>>> int("hello")
Traceback ...ValueError: ...
```