Python 04

Elias Wachmann

2024

Crash course is over...

So far we've learned:

- How to setup Python & VS-code
- numpy, matplotlib, random numbers and basic file I/O
- Functions, files and how to fit data

Now we will start from the ground up with Python:

- Data types & operators
- Branching
- Loops ...

Content

Data types & Operators

Primitive types

- `int` for integers (1, 2, 3, ...)
- `float` for floating point numbers (1.0, 3.1415, ...)
- `double` for double precision floating point numbers (1.0, 3.1415, ...)
- `bool` for booleans (True, False)
- `str` for strings ('Hello', 'World', ...)
- `char` for characters ('a', 'b', ...)
- `None` for null values (None)

numpy - data types

Non-exhaustive list of numpy data types:

- `int8, int16, int32, int64` for integers
- `uint8, uint16, uint32, uint64` for unsigned integers
- `float16, float32, float64, float128` for floating point numbers
- `complex64, complex128, complex256` for complex numbers
- `bool` for booleans
- `str` for strings
- `None` for null values

Types

- Python is dynamically typed
- Types are inferred from the value
- Types can be changed
  - `a = 1`
  - `a = 'a'`
- Types can be checked with `type(a)`

Operators

- Arithmetic operators
  - + addition (x + y)
  - - subtraction (x - y)
  - * multiplication (x * y)
  - / division (float) (x / y)
  - // division (int) (x // y)
  - ** exponentiation (x ** y)
  - % modulo (x % y)

Assignment Operators

- Assignment operators
  - = assigns a value to a variable (x = 5)
  - += adds a value to a variable (x += 3 == x = x + 3)
  - -= subtracts a value from a variable (x -= 3)
  - *= multiplies a variable (x *= 3)
  - /= divides a variable (x /= 3)
  - ... and so on

Comparison Operators

- Comparison operators
  - == equal ($x \equiv y$)
  - != not equal ($x \neq y$)
  - > greater than ($x > y$)
  - < less than ($x < y$)
  - >= greater than or equal to ($x \geq y$)
  - <= less than or equal to ($x \leq y$)

Logical Operators

- Logical operators
  - `and` returns True if both statements are true
    (x > 3 `and` x < 10)
  - `or` returns True if one of the statements is true
    (x > 3 `or` x < 4)
  - `not` returns False if the result is true
    (`not`(x > 3 `and` x < 10))

Identity Operators

- Identity operators
  - `is` returns True if both variables are the same object
    (x `is` y)
  - `is not` returns True if both variables are not the same object
    (x `is not` y)

Bitwise operators

- Bitwise operators
  - & AND (x & y)
  - | OR (x | y)
  - ^ XOR (x ^ y)
  - ~ NOT (~ x)
  - << left shift (x << 2)
  - >> right shift (x >> 2)

Conversion to other types

- `int(x)` converts x to an integer
- `float(x)` converts x to a float
- `str(x)` converts x to a string
- `bool(x)` converts x to a boolean
- `list(x)` converts x to a list
- `tuple(x)` converts x to a tuple
- `set(x)` converts x to a set
- `dict(x)` converts x to a dictionary
- `complex(x)` converts x to a complex number
- `bytes(x)` converts x to a bytes object
- `bytearray(x)` converts x to a bytearray object

Precision Issues in Floating Point Arithmetic

In many programming languages, especially those using floating point arithmetic, **0.1 + 0.1** might not always equal **0.2**.

- This is due to the inherent limitations of representing real numbers in binary format.
- Floating point numbers have limited precision, and some numbers cannot be represented exactly.
- Therefore, arithmetic operations involving floating point numbers may introduce rounding errors.

For example, in Python:

- `>>> 0.1 + 0.1` yields `0.2` (as expected).
- However, `>>> 0.1 + 0.1 + 0.1` yields `0.30000000000000004`.

Similar issues may occur in other programming languages.

Lists & Tuples

Lists

Lists are a collection of items that are **ordered** and **changeable**.

Example:

```
1 numbers = [1, 2, 3, 4]  # Create a list
```

Note: lists are **mutable** objects. Meaning that they can be changed after they have been created.

Lists

Example:

```
1  numbers = [1, 2, 3, 4]  # Create a list
2  print(numbers)
3  print(numbers[0])  # Output: 1
4  numbers[0] = 10  # Change the first element
5  print(numbers)  # Output: [10, 2, 3, 4]
6  numbers.append(6)  # add 6 to the end
7  print(numbers)  # Output: [10, 2, 3, 4, 6]
8  numbers.remove(3)  # remove 3
9  print(numbers)  # Output: [10, 2, 4, 6]
10 print(len(numbers))  # Output: 4
```

Tuples

Tuples are a collection of items that are **ordered** and **unchangeable**.

Example:

```
1 numbers = (1, 2, 3, 4)  # Create a tuple
```

Note: tuples are **immutable** objects. Meaning that they can **not** be changed after they have been created.

Tuples

Example:

```
1 numbers = (1, 2, 3, 4)  # Create a tuple
2 print(numbers)
3 print(numbers[0])  # Output: 1
4 print(len(numbers))  # Output: 5
5 more_numbers = (6, 7, 8)
6 all_numbers = numbers + more_numbers
7 print(all_numbers)  # (1, 2, 3, 4, 6, 7, 8)
8 a, b, c = numbers
9 print(a, b, c)  # Output: 1 2 3
```

Branching

If, elif, else

If, elif, else statements are used to execute code depending on a condition.

- `if` is used to execute code if a condition is true.
- `elif` is used to execute code if a condition is true and the previous conditions were false.
- `else` is used to execute code if all previous conditions were false.

If, elif, else

Caution: `if`, `elif` and `else` statements must be indented. This way the python-interpreter knows which code belongs to the `if`, `elif` or `else` statement.

```
1 x = 42
2 if (x > 0):
3     print("x is positive")
4 elif (x == 0):
5     print("x is zero")
6 else:
7     print("x is negative")
```

Using If on lists or strings

You can use `if` statements on lists or strings with the `in` / `any` / `all` operator.

```
my_list = [-1, 0, 1, 10, 42, 5]
if 0 in my_list:
    print("0 is in the list")
if all(x > 0 for x in my_list):
    print("all elements are positive")
if any(x > 0 for x in my_list):
    print("at least one element is positive"
    )
```

If with multiple conditions

You can use `and` & `or` to combine multiple conditions.

```
x, y = 5, True
if x > 0 and y:
    print("x is positive and y is True")
elif x > 0 or y:
    print("x is positive or y is True")
if not y:
    print("y is False")
```

The `not` keyword is used to invert the result of a condition.

Ordering of conditions

```
1 x = 42
2 if x > 0:
3     print("x is positive")
4 elif x == 42:
5     print("x is 42")
6 else:
7     print("x is <= 0")
```

What is the output of this script?

Ordering of conditions

```
1 x = 42
2 if x > 0:
3     print("x is positive")
4 # elif not triggered because x > 0 is True
5 elif x == 42:
6     print("x is 42")
7 else:
8     print("x is <= 0")
```

`x is positive` will be printed.