Python Programs and Computing Expressions

Programming with Python (for Bioinformatics)

Sven Rahmann

Summer 2024

Two Ways to Run Python

Interactively

Start the Python interpreter on the command line:

`$ python`

```
Python 3.10.6 | packaged by conda-forge | (main, Aug 22 2022, ...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt `>>>`, you may enter Python statements or expressions interactively, and directly see their results displayed in the terminal.
This is called a **REPL** (read-eval-print loop).

Two Ways to Run Python

Interactively

Start the Python interpreter on the command line:

`$ python`

```
Python 3.10.6 | packaged by conda-forge | (main, Aug 22 2022, ...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt >>>, you may enter Python statements or expressions interactively, and directly see their results displayed in the terminal.
This is called a **REPL** (read-eval-print loop).

Run Python Programs (Scripts)

`$ python myprogram.py`

This will execute the statements of `myprogram.py` and return to the terminal.

Python Programs

- Python programs are sequences of **statements**.
- There are many types of statements; all will be discussed soon.
  - **expressions**
  - assignments (=), function and class definitions (`def`, `class`)
  - conditionals: `if ... elif ... else`
  - loops: `for`, `while`
  - context managers: `with`
  - and many more
- In a program, the interpreter executes one statement after another.
- In a REPL, the interpreter executes one statement and shows the result.

Interpretation vs. Compilation

- C or C++ is a compiled language, i.e. translated to machine code before execution.
- Python is an interpreted language (actually, a hybrid).
  The interpreter looks at each statement as it arrives.

Object-Orientation

- Python is **object-oriented**. This means: **Everything** is an object.
- An object has both **attributes** (data) and **methods** (code).
- Attributes are for storing data and state, i.e. remembering things.
- Methods are for acting (on other objects), i.e. doing stuff.
- In fact, methods are attributes that can be called.

Object-Orientation

- Python is **object-oriented**. This means: **Everything** is an object.
- An object has both **attributes** (data) and **methods** (code).
- Attributes are for storing data and state, i.e. remembering things.
- Methods are for acting (on other objects), i.e. doing stuff.
- In fact, methods are attributes that can be called.

Example

- **Everything** is an object. For example, the integer number `1`.
- Two of its attributes are its **real** and its **imaginary** part:
  `((1).real, (1).imag)` results in `(1,0)`
- One of its methods caluclates how many bits we need to represent it:
  `(1).bit_length()` results in 1; one bit suffices.
- The **dot operator** (`.`) accesses both attributes and methods of an object.
- Methods have to be **called** with parentheses `(...)`: `(1).bit_length` gives
  `<built-in method bit_length of int object at 0x7fd88fdcb930>`.

Types

Every object in Python has a **type**. It defines which attributes and methods an object x has, and can be seen with `type`(x).

Basic types

- `int`: represents an integer number, arbitrary precision
- `float`: represents a floating point number (64 bits)
- `bool`: represents a logical value (`True` or `False`),
- `str`: represents a string or text (needs quotes)
- `NoneType`: the type of `None` (special unique object for nothing)

Types

Every object in Python has a **type**. It defines which attributes and methods an object x has, and can be seen with `type`(x).

Basic types

- `int`: represents an integer number, arbitrary precision
- `float`: represents a floating point number (64 bits)
- `bool`: represents a logical value (`True` or `False`),
- `str`: represents a string or text (needs quotes)
- `NoneType`: the type of `None` (special unique object for nothing)

Examples

```
>>> type(1)  # <class 'int'>
>>> type(1.0)  # <class 'float'>
>>> type(True)  # <class 'bool'>
>>> type("1"), type('one'), type("""Eins""")  # <class 'str'>
>>> type(None)  # <class 'NoneType'>
```

Operators

- Operators operate (act) on objects. But so do methods!

Operators

- Operators operate (act) on objects. But so do methods!
- Operators are just a certain type of methods (with special syntax).
- Many operators are binary: They combine two objects into a new one.
- Consider `1 + 2`: Here `1` is an object, `2` is an object, and `+` is an operator. The result, `3`, is again an object (different from `1` and `2`).
- In fact, the `+` operator results in a **method call**: `1 + 2` is the same as `(1).__add__(2)` (or sometimes `(2).__radd__(1)`)
- Methods with **double underscores** ("dunder") are special or **magic methods**; in particular, every operator corresponds to a **dunder method**.

Operators

- Operators operate (act) on objects. But so do methods!
- Operators are just a certain type of methods (with special syntax).
- Many operators are binary: They combine two objects into a new one.
- Consider `1 + 2`: Here `1` is an object, `2` is an object, and `+` is an operator. The result, `3`, is again an object (different from `1` and `2`).
- In fact, the `+` operator results in a **method call**: `1 + 2` is the same as `(1).__add__(2)` (or sometimes `(2).__radd__(1)`)
- Methods with **double underscores** (“dunder”) are special or **magic methods**; in particular, every operator corresponds to a **dunder method**.

Popular operators

```
+ - * / // % ** >> << @ . [] ()
== < > <= >= != in not in is is not
and or not & | ~ ^
... if ... else ...    :=
```

Expressions

Using objects and operators (and functions),
we can build and evaluate arbitrarily complex **expressions**.

Expressions

Using objects and operators (and functions),
we can build and evaluate arbitrarily complex **expressions**.

A painter buys material for 3000 EUR, has transportation costs of 40 EUR, works many hours for 4200 EUR. To his invoice, he adds 19% value added tax. For his best customer, he gives a 2% rebate on the whole sum and rounds **down** to the nearest integer.

Expressions

Using objects and operators (and functions),
we can build and evaluate arbitrarily complex **expressions**.

A painter buys material for 3000 EUR, has transportation costs of 40 EUR, works many hours for 4200 EUR. To his invoice, he adds 19% value added tax. For his best customer, he gives a 2% rebate on the whole sum and rounds **down** to the nearest integer.

```
>>> (3000 + 40 + 4200) * 1.19 * (1 - 0.02)
8443.288
>>> int((3000 + 40 + 4200) * 1.19 * (1 - 0.02))  # truncate to integer
8443
```

What are expressions?

Note

We here define well-parenthesized expressions.
Many parentheses can be removed once we agree on **operator precedence**.

Base case

Any object (by itself) is an expression.

What are expressions?

Note

We here define well-parenthesized expressions.
Many parentheses can be removed once we agree on **operator precedence**.

Base case

Any object (by itself) is an expression.

Inductive cases

If x, y, c are expressions, then

- $(\triangle x)$ is an expression, where $\triangle$ is a unary operator, like `~` (negation) or `-` (unary minus);
- $(x \,\square\, y)$ is an expresion, where $\square$ is a binary operator, like `+`, `-`, `*`, `/`, `//`, `%`, `**`, etc.;
- (x `if` c `else` y) is an expression (ternary operator).

What are expressions?

More inductive cases

If f is a function that takes n arguments, and $x_1, \ldots, x_n$ are expressions, then

- $f(x_1, x_2, \ldots, x_n)$ is an expression.

If f is a method that takes n arguments, and x and $y_1, \ldots, y_n$ are expressions, then

- $x.f(y_1, \ldots, y_n)$ is an expression.

What are expressions?

More inductive cases

If f is a function that takes n arguments, and $x_1, \ldots, x_n$ are expressions, then

- $f(x_1, x_2, \ldots, x_n)$ is an expression.

If f is a method that takes n arguments, and x and $y_1, \ldots, y_n$ are expressions, then

- $x.f(y_1, \ldots, y_n)$ is an expression.

Note

There are more types of expressions, which will be covered later, e.g.

- list, dict, set constructions
- list, dict, set comprehensions
- generator expressions

Examples

- Multiplying a string: five times “abc”?

```
>>> 5 * "abc"    # same as 'abc' * 5
"abcabcabcabcabc"
```

Examples

- Multiplying a string: five times “abc”?

```
>>> 5 * "abc"    # same as 'abc' * 5
"abcabcabcabcabc"
```

- Adding two numbers vs. two strings

```
>>> 11 + 11,  "11" + "11"
22, "1111"
```

Examples

- Multiplying a string: five times “abc”?

```
>>> 5 * "abc"    # same as 'abc' * 5
"abcabcabcabcabc"
```

- Adding two numbers vs. two strings

```
>>> 11 + 11,  "11" + "11"
22, "1111"
```

- How many bits for the number that consists of 17 1s ?

```
>>> int(17 * "1").bit_length()
54
>>> bin(int(17 * "1"))
'0b100111011110010111111100100110110101100111000111000111'
```

Examples

- Multiplying a string: five times "abc"?

```
>>> 5 * "abc"    # same as 'abc' * 5
"abcabcabcabcabc"
```

- Adding two numbers vs. two strings

```
>>> 11 + 11,  "11" + "11"
22, "1111"
```

- How many bits for the number that consists of 17 1s ?

```
>>> int(17 * "1").bit_length()
54
>>> bin(int(17 * "1"))
'0b100111011110010111111001001101101011001110001110001111'
```

Operators (like +, ∗) act differently, depending on the type.

Operator Descriptions

- +: addition (numbers), concatenation (strings, sequences)
- −: subtraction (numbers), difference (sets, Counter)
- *: multiplication (numbers), repetition (number and string)
- /: true division (numbers)
- //: integer division (integers)
- %: remainder after integer division (integers), “mod”
- **: exponentiation, e.g. `3 ** 4` is 81.
- >>: shift right (integers)
- <<: shift left (integers)
- @: special multiplication (e.g. for matrices), used in libraries

- &: bitwise and (integers)
- |: bitwise or (integers)
- ^: bitwise xor (integers)
- ~: bitwise negation (integers), `~a == (-a) - 1`

Operator Descriptions

Boolean operators, return `True` or `False`

- `==`, `!=`: is equal?, is unequal?
- <, >: is smaller?, is larger?
- <=, >=: is less or equal?, is greater or equal?
- `in`: test for containment, e.g. string inside string (`'ab' in 'xyabuv' == True`)
- `not in`: negation of `in`
- `is`, `is not`: test for object identity

- `and`: logical and (of two conditions)
- `or`: logical or
- `not`: logical negation

Built-in Functions

- `print`$(x, y, z, \dots)$: print a string representation of the arguments to the screen
- `max`$(x, y, z, \dots)$: return the maximum of all arguments
- `min`$(x, y, z, \dots)$: return the minimum of all arguments
- `abs`(x): return the absolute value of x
- `sum`$((x, y, z, \dots))$: return the sum of the iterable, note double parentheses!
- `pow`(b, x, m): return b^x, or b^x mod m (if m is given)
- `len`(x): return the length of x (when it makes sense)
- `type`(x): return the type of x

Built-in Functions

- `print`$(x, y, z, \ldots)$: print a string representation of the arguments to the screen
- `max`$(x, y, z, \ldots)$: return the maximum of all arguments
- `min`$(x, y, z, \ldots)$: return the minimum of all arguments
- `abs`(x): return the absolute value of x
- `sum`$((x, y, z, \ldots))$: return the sum of the iterable, note double parentheses!
- `pow`(b, x, m): return b^x, or $b^x \bmod m$ (if m is given)
- `len`(x): return the length of x (when it makes sense)
- `type`(x): return the type of x

Type conversions

- `int`(x): try to convert x (e.g., a string) to integer
- `float`(x): try to convert x (e.g., a string) to integer
- `str`(x): convert x (e.g., a number) to a string
- `bool`(x): return a boolean representation (`True`, `False`) of x

Math Functions

Modules and Namespaces

Python comes with “batteries included”, i.e. a lot of functionality.

The features are organized into different **modules**.

To access a feature, the corresponding module must be **imported**.

The features are then available in their own **namespace**.

Math Functions

Modules and Namespaces

Python comes with "batteries included", i.e. a lot of functionality.
The features are organized into different **modules**.
To access a feature, the corresponding module must be **imported**.
The features are then available in their own **namespace**.

Mathematical functions are in the `math` module and namespace.

```
>>> import math
>>> math.log(math.e)  # ln(e) = 1
>>> 2 * math.pi * 10.0  # circumference of a circle with radius 10
```

Math Functions

Modules and Namespaces

Python comes with "batteries included", i.e. a lot of functionality.
The features are organized into different **modules**.
To access a feature, the corresponding module must be **imported**.
The features are then available in their own **namespace**.

Mathematical functions are in the `math` module and namespace.

```
>>> import math
>>> math.log(math.e)  # ln(e) = 1
>>> 2 * math.pi * 10.0  # circumference of a circle with radius 10
```

It can be convenient to import **everything** from a module into our namespace.
This saves typing, but can be a source of confusion.

```
>>> from math import *  # use with caution!
>>> log(e),  2 * pi * 10.0  # no need to type math.
```

Examples

- The absolute value of the minimum of the numbers $1, -5, 7, -11, 667, -3$

```
>>> abs(min(1, -5, 7, -11, 667, -3))
11
```

Examples

- The absolute value of the minimum of the numbers $1, -5, 7, -11, 667, -3$

```
>>> abs(min(1, -5, 7, -11, 667, -3))
11
```

- Computing large factorials is easy because Python has arbitrary-precision integers

```
>>> from math import *
>>> factorial(10)  # this is math.factorial
>>> factorial(100)  # wow!
```

Examples

- The absolute value of the minimum of the numbers $1, -5, 7, -11, 667, -3$

```
>>> abs(min(1, -5, 7, -11, 667, -3))
11
```

- Computing large factorials is easy because Python has arbitrary-precision integers

```
>>> from math import *
>>> factorial(10)  # this is math.factorial
>>> factorial(100)  # wow!
```

- From a set of 10 cards, you can choose 3.
  In how many different ways is this possible?

Examples

- The absolute value of the minimum of the numbers $1, -5, 7, -11, 667, -3$

```
>>> abs(min(1, -5, 7, -11, 667, -3))
11
```

- Computing large factorials is easy because Python has arbitrary-precision integers

```
>>> from math import *
>>> factorial(10)  # this is math.factorial
>>> factorial(100)  # wow!
```

- From a set of 10 cards, you can choose 3.
  In how many different ways is this possible?
  Answer: $\binom{10}{3} = 10!/(7! \cdot 3!)$

```
>>> import math
>>> math.comb(10, 3)  # comb for combinations
```

Examples

- The absolute value of the minimum of the numbers $1, -5, 7, -11, 667, -3$

```
>>> abs(min(1, -5, 7, -11, 667, -3))
11
```

- Computing large factorials is easy because Python has arbitrary-precision integers

```
>>> from math import *
>>> factorial(10)  # this is math.factorial
>>> factorial(100)  # wow!
```

- From a set of 10 cards, you can choose 3.
  In how many different ways is this possible?
  Answer: $\binom{10}{3} = 10!/(7! \cdot 3!)$

```
>>> import math
>>> math.comb(10, 3)  # comb for combinations
```

Please read the **math module documentation**.