An introduction to Python

Object-oriented programming
Iteration
Generators

Object-oriented programming

data+code->objects

- OOP/OOD
- Classes
- Inheritance

Procedural programming

In traditional *procedural* programming, there are

- data
- functions working on data

```
>>> m1 = create_matrix(10, 20)

>>> fill_matrix(m1, 2.3)

>>> write_matrix(m1, “m1.raw”)
```

Object-oriented programming

In *object-oriented* programming, data and functions working on data are tied on the same *class*:

```
>>> m1 = Matrix(10, 20)

>>> m1.fill(2.3)

>>> m1.tofile(“m1.raw”)
```

Classes

In *object-oriented* languages, every object is an *instance* of a particular *class*.

The class completely defines the object's type, so we'll refer indifferently to the object's type or to the object's class.

A class do not simply define the object's content: it defines also the objects *algebra*, i.e. all the operation supported by objects.

Design: interface and implementation

In (object-oriented) design it’s a good idea to fix first the interface and then the implementation.

Remember:

Code to an interface, not to an implementation.

The `Fraction` class

We want to design and implement a `Fraction` class for fractions.

This is only for exercise: the standard library already has a `fractions` module implementing this class.

A simple class

Expected interface:

```
>>> fr = Fraction(2, 3)
>>> print("{}/{}".format(fr.numerator, fr.denominator))
2/3
```

Implementation:

```
class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
```

The `__init__` method

The `__init__()` method is the class initializer; it is called when the *Fraction* object is created.

The first argument is `self`, the instance to be initialized; this is the first argument for all the methods.

The initializer adds two attributes (*numerator* and *denominator*) to `self`.

Technically it's not a constructor, since the `self` object already exists.

The actual constructor is `__new__()` and is a special method (a class method). Normally there is no need to define it.

The `numerator` and `denominator` attributes

By default instance attributes are public:

```
>>> f0 = Fraction(5, 2)
>>> f0.numerator
5
>>> f0.numerator = 7
>>>
```

The `simplify()` method

We now want to add a method that can be used to simplify the fraction.

The interface we want to implement is

```
>>> f0 = Fraction(15, 6)
>>> print("{}/{}".format(f0.numerator, f0.denominator))
15/6
>>> f0.simplify()
>>> print("{}/{}".format(f0.numerator, f0.denominator))
5/2
```

The `simplify()` method

```
class Fraction(object):

    ...

    def simplify(self):

        while True:

            for n in range(2, self.numerator - 1):

                if self.numerator % n == 0 and self.denominator % n == 0:

                    self.numerator //= n

                    self.denominator //= n

                    break

            else:

                break
```

String conversion

It would be better to write simply

```
>>> print(f0)
```

instead of repeating the following code to print a fraction

```
>>> print("{}/{}".format(f0.numerator, f0.denominator))
```

All the Python objects can be converted to strings, so also our fractions can be printed; anyway, the default string conversion is not meaningful:

```
>>> print(f0)
<__main__.Fraction object at 0x7fc854692940>
>>>
```

String conversion

What we would like to implement is the following interface:

```
>>> f0 = Fraction(15, 6)
>>> print(f0)
(15/6)
```

We need to define the `__str__()` method. It must return a string.

The `__str__()` method

```
class Fraction(object):
    ...
    def __str__(self):
        return "({}/{})".format(self.numerator, self.denominator)
```

Example:

```
f0 = Fraction(15, 6)
>>> print(f0)
(15/6)
```

Object's representation

For the same reason, it would be better to change the default representation for fractions. It should be the Python source code that can be used to construct that object, so a good interface is:

```
>>> f0 = Fraction(15, 6)

>>> print(repr(f0))

Fraction(15, 6)
```

We need to define the `__repr__()` method. It must return a string.

The `__repr__()` method

```
class Fraction(object):
    ...
    def __repr__(self):
        return "{t}({n}, {d})".format(
            t=type(self).__name__,
            n=self.numerator, d=self.denominator)
```

Usage:

```
>>> f0
Fraction(15, 6)
```

Design: class invariants

A *class invariant* is a particular condition or property that the class instances must have. **The entire public interface of the class must satisfy the class invariants.**

In our case, we want that <u>every fraction is always simplified</u>.

Invariants are preserved only in the public interface: they are generally temporarily broken inside method.

The simplification invariant

We want to force that every `Fraction` object is automatically simplified:

```
>>> print(Fraction(15, 6))
(5/2)
```

We can force simplification during construction by calling the `simplify` method at the end of the initializer:

```
class Fraction(object):
    def __init__(self, numerator, denominator):
        ...
        self.simplify()
```

Is this invariant broken?

The invariant is broken if the public interface allows to obtain a fraction that is not simplified. Unfortunately this is possible, since we gave public access to the *numerator* and *denominator* attributes:

```
>>> f0 = Fraction(15, 6)
>>> f0
Fraction(5, 2)
>>> f0.numerator = 10
>>> f0
Fraction(10, 2)
```

How to fix this problem?

Generally classes do not give public access to their *innards* (the *representation*); attributes are generally **private**. Helper functions can be added to give read-only access to the attributes.

Attributes whose name starts with a double underscore are **private**.

So we can change the implementation of the class by changing `self.numerator` with `self.__numerator`, and `self.denominator` wit `self.__denominator`.

Generally a single underscore is used. Attributes starting with underscore are considered **protected**.

Private members

```
class Fraction(object):
    def __init__(self, numerator, denominator):
        self.__numerator = numerator
        self.__denominator = denominator
        self.simplify()

    def numerator(self):
        return self.__numerator
```

Private/protected methods

As attributes, methods starting with double underscore are private, and methods starting with a single underscore are *considered* protected.

In this case, since the simplification is an invariant, the `simplify` method is useless in the public interface. It's better to avoid useless public functions or attributes, so we will protect this method.

We choose to have a protected, and not a private method, since using this method is useless, but not dangerous.

```
class Fraction(object):
    def __init__(self, numerator, denominator):
        ...
        self._simplify()
```

Read-only property

As an alternative, a read-only property can be used. A property is an attribute-like object: it is used as an attribute, but implemented as a member.

```
class Fraction(object):
    ...
    @property
    def numerator(self):
        return self.__numerator
```

Properties are by default *read-only* attributes.

Read-only property

In this case, the public interface allows accessing numerator as a read-only member:

```
>>> f0.numerator
5
>>> f0.numerator = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Read-only property

Properties can be used not only to “protect” attributes, but also to create “virtual” attributes. For instance

```
>>> print(f0)
Fraction(5, 2)
>>> print(f0.value)
2.5
>>>
```

We can implement this interface without introducing a new attributes (depending on the other).

Read-only property

Properties can be used not only to “protect” attributes, but also to create “virtual” attributes. For instance

```
class Fraction(object):
    ...
    @property
    def value(self):
        return self.__numerator / self.__denominator
```

Property setter

Properties allow also the set operation. Consider this interface:

```
>>> f1 = Fraction(5, 6)
>>> print(f1)
(5/6)
>>> f1.numerator +=5
>>> print(f1)
(3/2)
```

Notice that the simplification invariant is satisfied after the numerator update. We can add a *property setter*.

Property setter

```
class Fraction(object):
    @property
    def numerator(self):
        return self.__numerator

    @numerator.setter
    def numerator(self, value):
        self.__numerator = value
        self._simplify()
```

The property *getter* function is used in read-only expressions such as:
`>>> print(fr.numerator)`

The property *setter* function is used in assignment expressions such as:
`>>> fr.numerator = 9`

Property setter

We will add a setter for the *numerator* and the *denominator* properties, but not for the *value*, since our interface does not need it.

Property deleter

Properties can also have a deleter, which is called when the property attribute is deleted, for instance:

```
>>> del obj.prop
```

If the property *prop* has deleter function, it is called; otherwise an error is raised.

In this case, we won’t define a deleter, since our interface don’t need it, and since it would be absurd to delete the numerator or the denominator.

Arithmetic

We now want to add arithmetic to our class. The interface we want to implement is the simplest one:

```
>>> f0 = Fraction(5, 2)
>>> f1 = Fraction(2, 3)
>>> f0 + f1
Fraction(19, 6)
>>>
```

The __add__ method

In order to allow addition, the __add__ method must be added:

```
class Fraction(object):
    ...
    def __add__(self, other):
        return Fraction(
            numerator=self.numerator * other.denominator + \
                      other.numerator * self.denominator,
            denominator=self.denominator * other.denominator)
```

Operator syntax

The operator syntax is only *syntactic sugar*: indeed operators are simply methods with a special name. What we write is

```
>>> f0 + f1
```

but what the interpreter executes is

```
>>> f0.__add__(f1)
```

Mixed-type arithmetic

It would be nice to be able to mix fractions and integers in arithmetic expressions, as we can do with float and int objects:

```
>>> f0 = Fraction(5, 2)

>>> print(f0 + 5)

(15/2)

>>> print(5 + f0)

(15/2)

>>>
```

The `__add__` method

The first mixed expression (`f0 + 5`) can be obtained by slightly changing the method:

```
def __add__(self, other):
    if not isinstance(other, Fraction):
        other = Fraction(other, 1)
    return Fraction(...)
```

The `isinstance(other, Fraction)` function returns True if `other` is a `Fraction` (introspection).

The `__radd__` method

The second mixed expression (`5 + f0`) can be obtained by adding a `__radd__` (reverse add) method.

Indeed, when an expression `obj0 + obj1` is found, the interpreter first tries to execute it as `obj0.__add__(obj1)`; if this is not possible, it then tries to execute `obj1.__radd__(obj0)`.

In our case the implementation of the `__radd__` method is simple, due to the commutative property of the addition:

```
    def __radd__(self, other):
        return self.__add__(other)
```

The `Matrix` class

We now want to design a `Matrix` class.

This is only an exercise: the `numpy` package contains a much better implementation.

The matrix interface

We want to implement the following interface:

```
>>> m0 = Matrix(4, 10, fill=1.0)
>>> print(m0[2, 3])
1.0
>>> m0[2, 3] = 5.5
>>> m1 = Matrix(4, 10, fill=2.0)
>>> m2 = m0 + m1
>>> m0 += m1
```

The `__getitem__()` method

In order to implement the item access, as in

```
>>> print(m0[2, 3])
```

we must implement the `__getitem__(...)` method:

```
class Matrix(object):
    ...
    def __getitem__(self, index):
        i, j = index
        return self.__data[i][j]
```

The `__setitem__()` method

In order to implement the item assignment, as in

```
>>> m0[2, 3] = 5.5
```

we must implement the `__setitem__(...)` method:

```
class Matrix(object):
    ...
    def __setitem__(self, index, value):
        i, j = index
        self.__data[i][j] = value
```

Mutable or immutable?

We already know how to implement a class with the *add* interface. But notice the last line:

```
>>> m0 += m1
```

We know that, for immutable types, this += operation creates a new object; `m0` becomes a symbolic name for that new object. Since a matrix can be very big, this is not memory efficient: it would be better to have a mutable type, and to change the matrix object *in-place*.

The `__iadd__()` method can be set to implement this in-place operator. It is defined as the `__add__()`; generally it returns the `self` object after it has been changed.

The __add__() method

```
def __add__(self, other):
    result = Matrix(self.__num_rows, self.__num_columns)
    for i in range(self.__num_rows):
        for j in range(self.__num_columns):
            result[i, j] = self[i, j] + other[i, j]
    return result
```

The __iadd__() method

```
def __iadd__(self, other):
    for i in range(self.__num_rows):
        for j in range(self.__num_columns):
            self[i, j] += other[i, j]
    return self
```

Class attributes

Sometimes we need to set a class attribute, i.e. an attribute which is exactly the same for all the instances of the class.

In this case, it can be set directly in the class body.

For instance, in a `Circle` class we need to define the value of `pi`.

Class attributes

```
class Circle(object):
    pi = 3.141592653589793

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return self.pi * self.radius ** 2
```

Class attributes

A class attribute belongs to the class, not to the instance. It can be accessed also through the class:

```
>>> print(Circle.pi)
3.141592653589793
>>>
```

Class methods

Sometimes we need to define a method whose behavior does not depend on the instance, but only on the class.

For instance, the Matrix class could implement a method to read the class from file. The file contains shape and data. The `from_file` method mustn't be applied to a constructed matrix, since a constructed matrix already has a shape (and data) that do not necessarily match the file content. It can be defined as class method.

The `matrix.to_file(...)` method

The `to_file(...)` method is a normal instance method, which can be applied to an already constructed instance:

```
def to_file(self, filename):
    with open(filename, "w") as f_out:
        f_out.write("{!r} {!r}\n".format(
            self.__num_rows, self.__num_columns))
        for i in range(self.__num_rows):
            for j in range(self.__num_columns):
                f_out.write(str(self[i, j]) + '\n')
```

The `matrix.from_file(...)` method

```
@classmethod
def from_file(cls, filename):
    with open(filename, "r") as f_in:
        lst = f_in.readline().strip().split()
        num_rows, num_columns = int(lst[0]), int(lst[1])
        instance = cls(num_rows, num_columns)
        for i in range(num_rows):
            for j in range(num_columns):
                instance[i, j] = float(f_in.readline().strip())
    return instance
```

`self` or `cls`

The first argument of an instance method is the instance itself; it is generally named `self`.

The first argument of a class method is the class itself; it is generally named `cls`.

Do not use other names!

Matrix I/O example

```
>>> m = Matrix(2, 10, fill=2.5)
>>> m.to_file("m.txt")
>>> m2 = Matrix.from_file("m.txt")
>>> print(m2)
2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5
2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5
>>>
```

Other magic methods

The list of magic `__methods__` is very long:

- `__delitem__()`: remove an item
- `__len__()`: return the object length
- `__iter__()`: iterate on the object
- **`__del__()`**: the finalizer; it's called just before the object is destroyed (normally there is no need to implement it)
- `__call__(*args, **kwargs)`: if defined, the object can be called as a function (it's a *functor*).

Operators and magic methods

Almost all the Python operators have corresponding magic methods, so they can be defined in classes:

- `a + b -> a.__add__(b)`
- `a - b -> a.__sub__(b)`
- `a * b -> a.__mul__(b)`
- `a / b -> a.__truediv__(b)`
- `a // b -> a.__foordiv__(b)`
- `a % b -> a.__mod__(b)`
- `divmod(a, b) -> a.__divmod__(b)`
- `...`

We can overwrite all this operators; anyway, **we cannot change their precedence**: the standard Python precedence will always be applied.

The class docstring

Also classes have docstrings.

```
>>> class Fraction(object):
...     """Fraction instances have numerator/denominator attributes"""
...
>>>
```

Inheritance

Inheritance is an essential feature in object-oriented programming.

A class can inherit from another one (its *base class* or *superclass*). In this case, the derived class inherits all the content (attributes and methods) of its superclass.

We make use of inheritance to

- Change an existing behavior
- Add new behaviors
- Enforce a common (abstract) interface to a hierarchy of classes

Change an existing behavior

We want to define a `rstr` class which behaves exactly as the builtin `str`, with the only exception that it is printed reversed:

```
>>> s = rstr("Hello, world!")
>>> print(s)
!dlrow ,olleH
>>>
```

A `rstr` is a `str` with a changed behavior.

Add new behaviors

Suppose we want to define a Path class to implement the path algebra:

```
>>> pdir = Path("/home/user/data")
>>> print(pdir)
/home/user/data
>>> print(pdir.dirname())
/home/user
>>> print(pdir.join("alpha", "beta", "gamma.txt"))
/home/user/data/alpha/beta/gamma.txt
```

Add new behaviors

A `Path` is a string with some more algebra. We can implement it by inheriting from the string class, and by defining the new behavior:

- `Path.dirname()`
- `Path.basename()`
- `Path.split()`
- `Path.join(*args)`
- `...`

Enforce a common interface

We know that many dict-like types exist:

- `dict`
- `defaultdict`
- `OrderedDict`
- `Counter`

They all have a common interface. We will introduce a new `frozendict` class in the dictionary hierarchy.

The “is a” property

The inheritance relationship can be expressed as “is a”:

- A `rstr` instance **is a** `str`
- A `Path` instance **is a** `str`
- A `Counter` instance **is a** `Mapping`

Changing an existing behavior: `rstr`

```
>>> class rstr(str):
...     def __str__(self):
...         return "".join(reversed(tuple(super().__str__())))
...
>>> s = rstr("Hello, world!")
>>> print(s)
!dlrow ,olleH
```

The `rstr` class

The base class is `str`, as stated in the first line. The `__str__` method is replaced by a new one. Notice that its implementation calls the original method. Here `super()` represents the superclass, so `super().__str__()` is the original str method.

Class hierarchies

The “is a” property is maintained in complex hierarchies:

```
>>> class Animal(object): pass

>>> class Mammal(Animal): pass

>>> class Cat(Mammal): pass

>>> wasabi = Cat()

>>> isinstance(wasabi, Animal)

True
```

The `object` class

All the Python classes inherit (directly or indirectly) from `object`. This is the base of the entire class hierarchy.

Adding a new behavior: Path

```
class Path(str):

    def dirname(self):

        return Path(os.path.dirname(self))



    def join(self, *parts):
        return Path(os.path.join(self, *parts))
    ...
```

The Path class

The base class is `str` also in this case.

Here `dirname()` is a new method, while `join()` is an overriden method.

The `frozendict` hierarchy

```
from collections import Mapping

class frozendict(Mapping):

    def __init__(self, *args, **kwargs):

        self.__dct = dict(*args, **kwargs)

    def __getitem__(self, key):

        return self.__dct[key]

    ...
```

The `frozendict` class

The base class is `collections.Mapping`. This is an abstract class enforcing the interface for an immutable mapping. When inheriting from this class, three methods must be defined:

- `__getitem__`
- `__len__`
- `__iter__`

Multiple inheritance

A class can have multiple base classes:

```
class Alpha(BaseA, BaseB, BaseC):
    pass
```

Iteration

The glue between data and algorithms

- The iterable/iterator duality

Iterable/iterator

An *iterable* is an object allowing iteration on it. To iterate means to traverse the object’s items, one item after the other.

An iterator is an object allowing iteration on an iterable.

- An object is *iterable* if its class has an `__iter__()` method returning an *iterator*.
- An object is an *iterator* if its class has a `__next__()` method returning the next iteration element.

iter()

The `iter()` function returns an iterator from an (iterable) object. It's a shorthand to call the object's `__iter__()` method.

```
>>> lst = [7, 11, 13]
>>> iterator = iter(lst)
>>> iterator
<list_iterator object at 0x7f5b56ebf9e8>
>>>
```

next()

The `next()` function moves the iterator forward and returns the next item, if any. It's simply a shorthand to call the object's `__next__()` method.

```
>>> next(iterator)
7
>>> next(iterator)
11
>>> next(iterator)
13
>>>
```

Python 2 vs Python 3

The name of the *next* method

`next()`

`__next__()`

StopIterator

The iteration process stops when the `__next__()` method raises a `StopIteration()` exception.

```
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

The `for` loop magic unveiled

We now understand how the for loop **`for item in sequence`** works:

- It creates an iterator to the sequence **`iterator = iter(sequence)`**
- It assigns **`item = next(iterator)`** (and then executes the loop's body) until **`next`** raises a `StopIteration` error

```
for item in lst:

    print(item)
```

```
iterator = iter(lst)

while True:

    try:

        print(next(iterator))

    except StopIteration:

        break
```

The *Fibonacci* iterable

We want to create an iterable object representing the Fibonacci sequence.

```
>>> for n in Fibonacci(limit=4):
...     print(n)
...
1
1
2
3
```

The limit argument is used to limit the number of generated fibonacci numbers; iteration must stop when the limit is reached.
This is only to show how to stop the iteration.

The `Fibonacci` class

The implementation of the `Fibonacci` class is quite simple:

```
class Fibonacci(object):
    def __init__(self, limit=None):
        self._limit = limit


    def __iter__(self):
        return FibonacciIterator(self._limit)
```

We now need to implement the `FibonacciIterator`.

The `FibonacciIterator.__init__` method

The `FibonacciIterator` initializer accepts the limit value and sets the internal iterator status:

```
class FibonacciIterator(object):
    def __init__(self, limit):
        self._limit = limit
        self._count = 0
        self._first, self._second = 1, 1
```

The status is needed to keep track of the current position in iteration and to save it between consecutive calls to the `__next__` method.

The `FibonacciIterator.__next__` method

The `FibonacciIterator` next method is responsible for

- Updating the internal status
- Returning the next iteration item
- Stopping the iteration when the limit is reached.

The FibonacciIterator.__next__ method

```
def __next__(self):
    self._count += 1
    if self._limit is not None and self._count > self._limit:
        raise StopIteration()
    if self._count <= 2:
        return 1
    else:
        self._first, self._second = self._second, self._first + self._second
        return self._second
```

Why two classes?

The iteration process involves two classes and two objects, the *iterable* and the *iterator*. Is that necessary?

For instance, consider a Range class which emulates the builtin `range` function.

A naive implementation with a single class is possible.

Range, single iterable/iterator class

A naive implementation with a single class:

```
class Range(object):
    def __init__(self, stop):
        self._stop = stop
        self._index = 0

    def __iter__(self):
        return self
```

```
    def __next__(self):
        if self._index >= \
            self._stop:
            raise StopIteration()
        value = self._index
        self._index += 1
        return value
```

Range, single iterable/iterator class

It seems to work:

```
>>> list(Range(6))
[0, 1, 2, 3, 4, 5]
>>>
```

Range, single iterable/iterator class

...but...

```
>>> r = Range(3)
>>> for i in r:
...     for j in r:
...         print(i, j)
...
0 1
0 2
>>>
```

The expected output is

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Range and RangeIterator classes

```
class Range(object):

    def __init__(self, stop):

        self._stop = stop


    def __iter__(self):

        return RangeIterator(self._stop)
```

```
class RangeIterator(object):

    def __init__(self, stop):

        Self._stop, self._index = stop, 0

    def __next__(self):

        if self._index >= self._stop:

            raise StopIteration()

        value = self._index

        self._index += 1

        return value
```

Range and `RangeIterator` classes

This works correctly, since two distinct iterators are used:

```
>>> for i in r:
...     for j in r:
...         print(i, j)
...
0 0
0 1
0 2
1 0
1 1 ...
```

Generator functions

Creating iterable objects is not difficult, anyway we need to implement two cooperative classes. Luckily, Python supports a simpler way to implement iteration: generator functions (**generators**).

Generator functions

A generator function is a function that can temporarily stop its execution, yield a value to the caller, and then be restarted by the caller; execution restarts just after the yield statement.

Generators use the `yield` statement instead of `return`.

The yield statement is a kind of “temporary” return: the execution of the function is suspended, and it’s up to the caller to resume it.

Generator functions

When a generator is called, it returns an *iterator*; the caller then uses this iterator as usual. Any call to next restart the execution of the generator function.

When the generator function returns, a `StopIteration` is automatically raised.

The `range_gen` generator

Consider this function:

```
>>> def range_gen(n):
...     i = 0
...     while i < n:
...         yield i
...         i += 1
...
>>>
```

The range_gen generator

```
>>> r = range_gen(3)
>>> r
<generator object range_gen at 0x7f039be9b150>
>>> next(r)
0
>>> next(r)
1
```

The `range_gen` generator

```
>>> next(r)
2
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Hands-on

Write a function that “returns” **all** the Fibonacci numbers, for instance to be used as follows:

```
>>> for n in fibonacci():
...     if is_pandigital(n):
...         print(n)
...         break
```

```
def is_pandigital(n):
    return len(set(str(n))) == 10
```

The `Fibonacci` generator

This is a solution:

```
>>> def fibonacci(*, first=1, second=1):
...     yield first
...     yield second
...     while True:
...         first, second = second, first + second
...         yield second
...
>>>
```

The `Fibonacci` generator

Notice that the `fibonacci()` iterator never stops! It's up to the caller to stop the iteration. For instance:

```
>>> for index, n in zip(range(10), fibonacci()):
...     print(index, n)
...
0 1
...
8 34
9 55
```

Generator expressions

Generator expressions have the list comprehension syntax with the generator semantics:

```
>>> [i ** 2 for i in range(5)]
[0, 1, 4, 9, 16]
>>> (i ** 2 for i in range(5))
<generator object <genexpr> at 0x7f06fc7181a8>
>>>
```

Summing all the squares of the first N integers

Suppose you want to sum all the squares of the first N integers:

```
>>> sum([i ** 2 for i in range(10000)])
333283335000
>>> sum([i ** 2 for i in range(100000)])
333328333350000
>>> sum([i ** 2 for i in range(1000000)])
333332833333500000
>>>
```

Summing all the squares of the first N integers

This wastes a lot of memory: indeed this sum does not really need a container. Using generator expressions only one integer is in the memory at any time:

```
>>> sum((i ** 2 for i in range(1000000)))
333332833333500000
>>>
```

When a generator expression is the only argument in a function call, parentheses can be omitted:

```
>>> sum(i ** 2 for i in range(1000000))
333332833333500000
>>>
```

Modules

How to organize Python source

- Modules
- Packages

Python modules

A Python module is simply a Python source file with `.py` extension. Modules are libraries: they can be imported and used in other modules or programs.

```
(python3.5)$ cat greetings.py

def greet(who):

    print("Hello, {}!".format(who))

def welcome(who):

    print("Welcome, {}!".format(who))
```

Importing modules

Modules can be imported:

```
>>> import greetings

>>> dir(greetings)

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'greet']

>>> greetings.greet("world")

Hello, world!

>>>
```

Importing from modules

Specific symbolic names can be imported from a module:

```
>>> from greetings import greet, welcome
>>> greet("world")
Hello, world!
>>> welcome("Guido")
Welcome, Guido!
>>>
```

Importing all from modules

It is possible to import all the symbolic names from a module:

```
>>> from greetings import *
>>> greet("world")
Hello, world!
>>> welcome("Guido")
Welcome, Guido!
>>>
```

This is not a good idea, since it pollutes the namespace!

The __all__ module attribute

Generally modules contain an `__all__` attribute, which is a list of the public symbolic names; when importing with *, only those names will be imported:

```
(python3.5)$ cat greetings.py
__all__ = [
    'greet',
    'welcome',
]
...
(python3.5)$
```

Module docstring

Modules have a docstring too; it’s a function string at the beginning of the file

```
(python3.5)$ cat primes.py
"""Functions to test primality"""


def is_prime(n):
    ...
(python3.5)$
```

Python packages

A Python package is a directory containing a module file `__init__.py` (the package initializer, which can be empty) and some other Python modules: for instance

```
number_theory/
    __init__.py
    primes.py
    divisors.py
```

Packages can contain subpackages.

Hands-on

Organize the Fibonacci exercise in a module.

Create a package containing the Fibonacci module and the divisors module.