An introduction to Python

Python data model
Exceptions
Decorators

Python data model

What is exactly a Python object?

- `__dict__`

A simple class

Consider this minimal class:

```
>>> class Simple(object):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...     def value(self):
...         return self.a + self.b
...
>>> s = Simple(2, 13)
```

The `__dict__` attribute

Every object has a __dict__ attribute:

```
>>> dir(s)

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'a', 'b']

>>> s.__dict__

{'a': 2, 'b': 13}

>>>
```

This dictionary contains the object’s attributes!

The `__dict__` instance attribute

Every object has a __dict__ attribute which contains the instance attributes.

The `__dict__` attribute

We know that classes are objects too. What is the `Simple.__dict__` attribute?

```
>>> Simple.__dict__

mappingproxy({'__doc__': None, '__init__': <function Simple.__init__ at
0x7f06fc719950>, '__module__': '__main__', '__dict__': <attribute '__dict__' of
'Simple' objects>, 'value': <function Simple.value at 0x7f06fc7199d8>,
'__weakref__': <attribute '__weakref__' of 'Simple' objects>})

>>>
```

It’s a little bit more complicated, anyway we can recognize a dict-like object containing the *value* method.

The `__dict__` class attribute

Every class has a __dict__ attribute which contains all the class members (attributes and methods).

Attribute access

When we get an instance attribute:

- It is first searched in the instance's `__dict__`
- If not found, it's searched in the class `__dict__`
- If not found, it's searched in the base classes `__dict__`

Garbage collection

The object’s lifetime

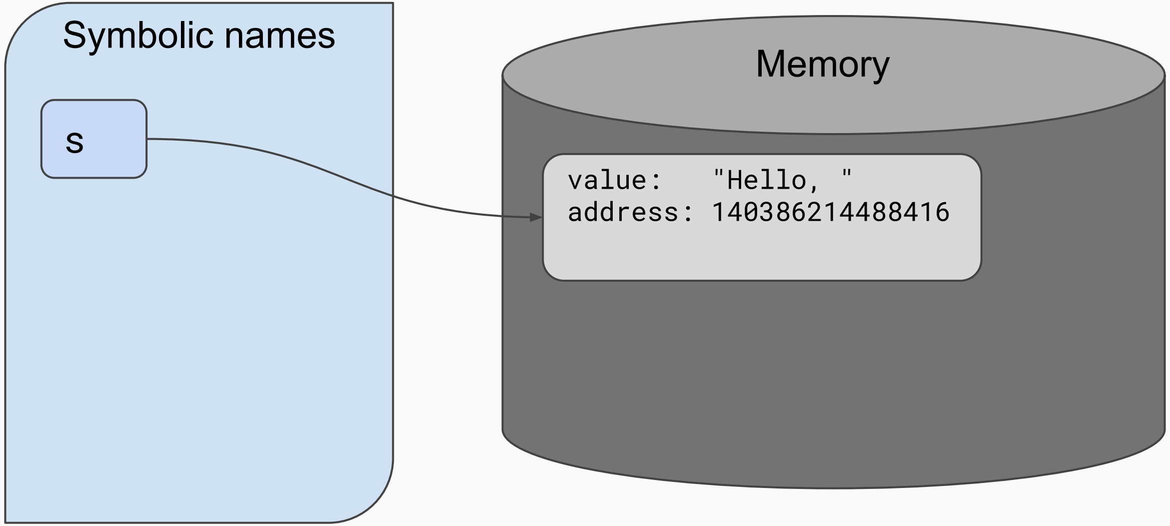
- Reference counting
- The garbage collector

Object reference

Remember how symbolic names are (temporarily) attached to objects.

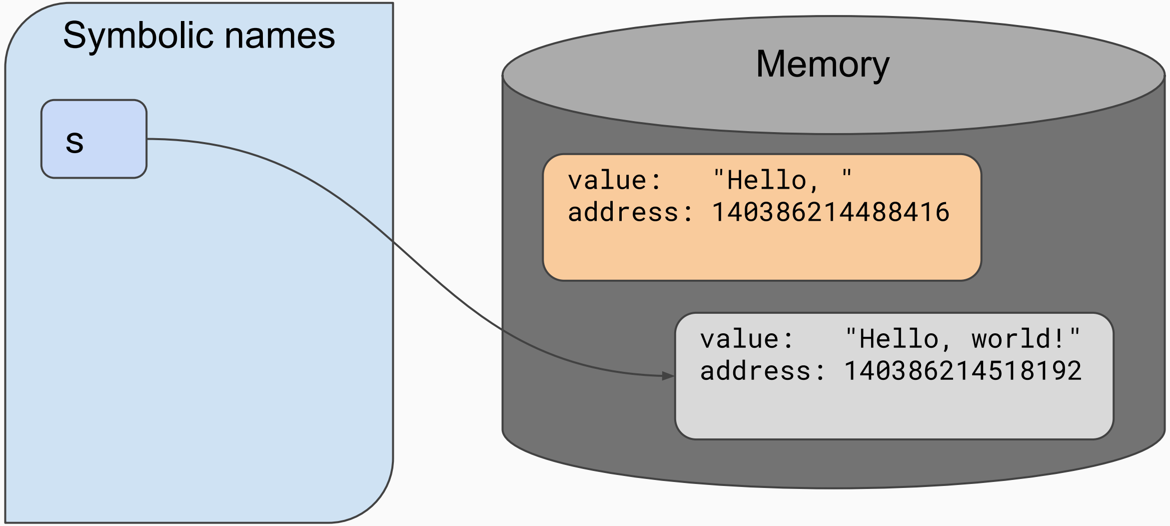
Object reference

```
>>> s = "Hello, "

>>> id(s)

140386214488416
```



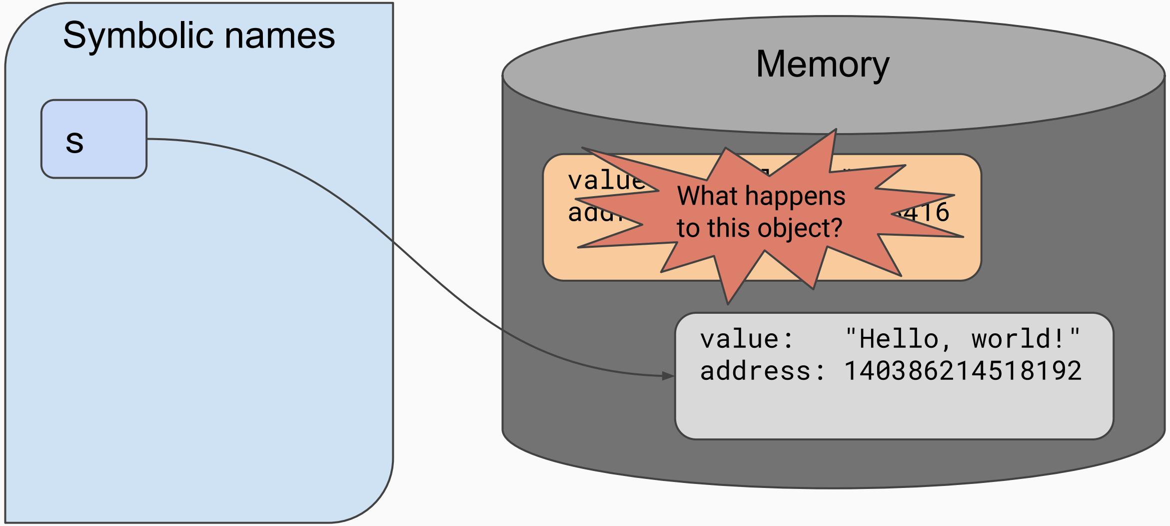
Object reference

```
>>> s = "Hello, "
>>> id(s)
140386214488416
>>> s += "world!"
>>> s
'Hello, world!'
>>> id(s)
140386214518192
```



Object reference

```
>>> s = "Hello, "
>>> id(s)
140386214488416
>>> s += "world!"
>>> s
'Hello, world!'
>>> id(s)
140386214518192
```



Object reference

In this example the object with *id* `140386214488416` is no longer in use. There is no way to use it, since no references are left to this object.

The *garbage collector* is responsible for destroying such unusable objects.

Reference counting **(CPython)**

Each Python object has a *reference count* attribute, which count the number of references.

This counter is increased, for instance:

- when a new symbolic name is given to the object;
- when the object is stored on a container;
- when another object refers to its through some attribute.

The *garbage collector* automatically deletes objects whose reference count becomes zero.

Reference counting **(CPython)**

The CPython interpreter (the one we are using) has a deterministic garbage collector, which tries to collect objects as soon as possible.

This is one of the most cpu-consuming interpreter activities.

Other Python implementation (pypy for instance) implements alternative garbage collection strategies, with a considerable speedup.

The garbage collector at work

Consider this class:

```
>>> class Trc(object):
...     def __init__(self):
...         print("INIT")
...     def __del__(self):
...         print("DEL")
...
>>>
```

The __del__ method is the finalizer: it is called just before the object is destroyed

The garbage collector at work

```
>>> obj = Trc()
INIT
>>> del obj
DEL
>>>
```

It seems we can decide when the object is destroyed. But it’s not our decision: it’s up to the garbage collector to destroy objects. We only can state that a reference to an object is no longer used.

The garbage collector at work

```
>>> obj = Trc()

INIT

>>> obj_alias = obj

>>> del obj

>>>
```

Why in this case the `__del__()` method has not been called? The reason is that **`del obj` deletes a reference, not an object!** In this case the `Trc` instance has another symbolic names referring to it, so the reference count is not zero.

The garbage collector at work

```
>>> obj = Trc()
INIT
>>> obj_alias = obj
DEL
>>> del obj
>>> obj_alias = 10
DEL
>>>
```

The garbage collector at work

```
>>> obj = Trc()
INIT
>>> obj_alias = obj
DEL
>>> del obj
>>> obj_alias = 10
DEL
>>>
```

Exceptions

The modern way to handle errors

- Exceptions
- Raising exceptions
- Caatching exceptions

Error handling

This is a list of the properties that we require to a modern language about error handling:

- Errors should never pass silently (the Zen of Python, n. 10)
- Error handling should not pollute code
- Errors can be handled partially

Error identification/handling

A very general property about errors:

- The code region where the error can easily be identified is not the code region where the error can be handled.

Error identification

For instance, an invalid argument value can be easily detected as an error by a third party library; nevertheless, this library function does not know what is the correct way to handle the error:

- stop the program? (generally it’s not a good idea…)
- use an alternate value? Which?
- ask the user for another value? How?

On the other hand, the caller function can easily handle this error after it has been detected.

Exceptions

Exceptions are raised when an error condition is detected. Consider this library function:

```
def invert(matrix):
    if not is_square(matrix):
        raise MatrixError("not a square matrix")
    det = compute_determinant(matrix)
    if det == 0:
        raise SingularMatrixError("singular matrix")
    ...
```

`raise`

When a `raise` statement is encountered, the function execution is stopped. Execution is then passed through the call stack to the first caller which is able to handle the raised exception.

Exception

An exception is an instance of an exception class; exception classes must inherit from `BaseException` (usually from `Exception`). Exceptions are usually organized hierarchically:

```
class MatrixError(Exception):
    pass
```

A `SingularMatrixError` error ***is a*** `MatrixError`!

```
class SingularMatrixError(MatrixError):
    pass
```

Exception

Exception classes have an error message (a string) as first argument. They can have optional arbitrary arguments.

Generally exception classes are empty; the exception type itself contains all the information needed to identify the error.

try/except

The user program calling this function can now check if an error is raised:

```
from third_party_matrix_library import load, invert
m = load("m.dat")
try:
    minv = invert(m)
except MatrixError:
    ...
```

`try/except`

If the execution of the *try* block raises a `MatrixError`, the execution stops and pass to the corresponding *except* block.

try/except

```
try:
    ...
except SingularMatrixError:
    ...
except MatrixError as err:
    ...
else:
    ...
finally:
    ...
```

This block is executed if a `SingularMatrixError` is raised

This block is executed if a `MatrixError`, but not a `SingularMatrixError`, is raised. Error is available as `err`.

This block is executed only if no errors are raised

This block is executed **always** after the *try* block

Partial error handling

Sometimes a caller function can handle the error only partially; in this case, a raise command without arguments in the except block will re-raise the catched error:

```
gui = create_gui()
try:
    ...
except SingularMatrixError:
    gui.close()  #  partial error handling (cleanup)
    raise
```

Decorators

How to decorate functions and classes

- Function decorators

Decorator pattern

To decorate a function means to implement a new function which is the original one, plus something executed before and/or after.

Timing

Suppose you often want to trace computing time for your functions. For each function, you have to

- start a timer before the function execution;
- execute the function;
- stop the timer at the end, and, for instance, print the elapsed time.

This *timing* recipe does not depend at all on the function's content.

Applying the same recipe to every possible function is tedious and error prone. Moreover, if you want to change the timing itself (for instance, you want to collect all the function call times and print a report at the end), you have to change each function.

The timed() function

```
def timed(function):
    def timed_function(*args, **kwargs):
        t0 = time.time()
        result = function(*args, **kwargs)
        print("elapsed: {:.2f} s".format(time.time() - t0))
        return result
    return timed_function
```

The `timed()` function

This function receives a `function` as argument.

It then creates a new function, `timed_function()`, which accept arbitrary positional and keyword arguments. The new function starts the times, calls the original `function`, shows the elapsed time, and returns the `function`'s return value.

This is possible because **functions are first-class citizens**, so they can be passed to functions or returned from functions.

The `timed()` function

Suppose you have this function and you want to time it:

```
def next_prime(n):
    p = n
    while not is_prime(p):
        p += 1
    return p
```

The `timed()` function

You can obtain a timed version of the function by calling `timed()`:

```
>>> from timed import timed
>>> from primes import next_prime
>>> timed_next_prime = timed(next_prime)
>>> timed_next_prime(10000000)
elapsed: 1.91 seconds
10000019
```

The `timed()` decorator

This `timed()` function is called a decorator: it takes a function as arguments and returns a decorated function.

The `@timed` syntax

Often you don't need the original function: you only need the decorated one. If you are defining your own function `compute()` and you want it to be timed, you can write:

```
>>> @timed
... def compute(f):
...     time.sleep(f)
...
>>> compute(0.35)
elapsed: 0.35 seconds
```

The `@decorator` syntax

The `@decorator` syntax is *syntactic sugar*: it only means that you want to replace the function you're defining with it's timed version.

```
>>> def compute(f):
...     time.sleep(f)
...
>>> compute = timed(compute)
>>>
```

The `@decorator` syntax

The `@decorator` syntax can be repeated in order to apply multiple decorators:

```
>>> @traced
... @timed
... @cached
... def compute(f):
...     pass
...
>>>
```

Decorators with arguments

Sometimes you need to pass an argument to the decorator:

```
>>> @timed("### {:.3f} seconds")
... def compute(f):
...     time.sleep(f)
...
>>> compute(0.35)
### 0.350 seconds
>>>
```

Decorators with arguments

```
def timed(fmt="elapsed: {:.2d} s"):
    def timed_decorator(function):
        def timed_function(*args, **kwargs):
            t0 = time.time()
            result = function(*args, **kwargs)
            print(fmt.format(time.time() - t0))
            return result
        return timed_function
    return timed_decorator
```