PYTHON: OOP

Basics

```
# class definition
class Vector:
  pass
```

Basics

```
# class definition
class Vector:
  pass

# instance creation
v = Vector()
```

Basics

```python
# class definition
class Vector:
  pass

# instance creation
v = Vector()

# you can add arbitrary attributes to instances
# (please don't)
v.a = 5
v.fn = lambda x: x + 1
```

Methods

```
class Vector:
  # special method called during instance creation
  # the first method parameter contains instance
  # (like C++ this)
  def __init__(self, x, y):
    self.x = x
    self.y = y
```

Methods

```
class Vector:
  # special method called during instance creation
  # the first method parameter contains instance
  # (like C++ this)
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def move(self, x, y):
    self.x += x
    self.y += y
```

Methods

```
class Vector:
  # special method called during instance creation
  # the first method parameter contains instance
  # (like C++ this)
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def move(self, x, y):
    self.x += x
    self.y += y

v = Vector(1, 2)
v.move(2, 4)  # == Vector.move(v, 2, 4)
v.x  # 3
```

Static attributes and methods

```
class Vector:
  x = 0  # 'static' attribute

  @staticmethod
  def zero():
    return Vector(0, 0)

v = Vector(1, 2)
v.x # 1
Vector.x # 0
```

What happens here?

```python
class Player:
  def __init__(self):
    self.position = Vector(0, 0)

def spawn_enemy_near_player(player):
  enemy_pos = player.position
  enemy_pos.move(10, 5)
  ...
```

Prefer immutable objects if possible

```python
class Player:
  def __init__(self):
    self.position = Vector(0, 0)

  def get_position(self):
    return Vector(self.position.x, self.position.y)

def spawn_enemy_near_player(player):
  enemy_pos = player.get_position().move(10, 5)
  ...
```

Prefer immutable objects if possible

```
class Player:
  def __init__(self):
    self.position = Vector(0, 0)

  def get_position(self):
    return Vector(self.position.x, self.position.y)

def spawn_enemy_near_player(player):
  enemy_pos = player.get_position().move(10, 5)
  ...
```

- Objects won't be changed out of nowhere

Prefer immutable objects if possible

```
class Player:
  def __init__(self):
    self.position = Vector(0, 0)

  def get_position(self):
    return Vector(self.position.x, self.position.y)

def spawn_enemy_near_player(player):
  enemy_pos = player.get_position().move(10, 5)
  ...
```

- Objects won't be changed out of nowhere
- Change detection is super easy (compare pointers)

Prefer immutable objects if possible

```
class Player:
  def __init__(self):
    self.position = Vector(0, 0)

  def get_position(self):
    return Vector(self.position.x, self.position.y)

def spawn_enemy_near_player(player):
  enemy_pos = player.get_position().move(10, 5)
  ...
```

- Objects won't be changed out of nowhere
- Change detection is super easy (compare pointers)
- Multithreading-friendly

Properties

```
class Vector:

  def length(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)
```

Properties

```
class Vector:

  def length(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)

v = Vector(1, 0)
x = v.length  # 1
```

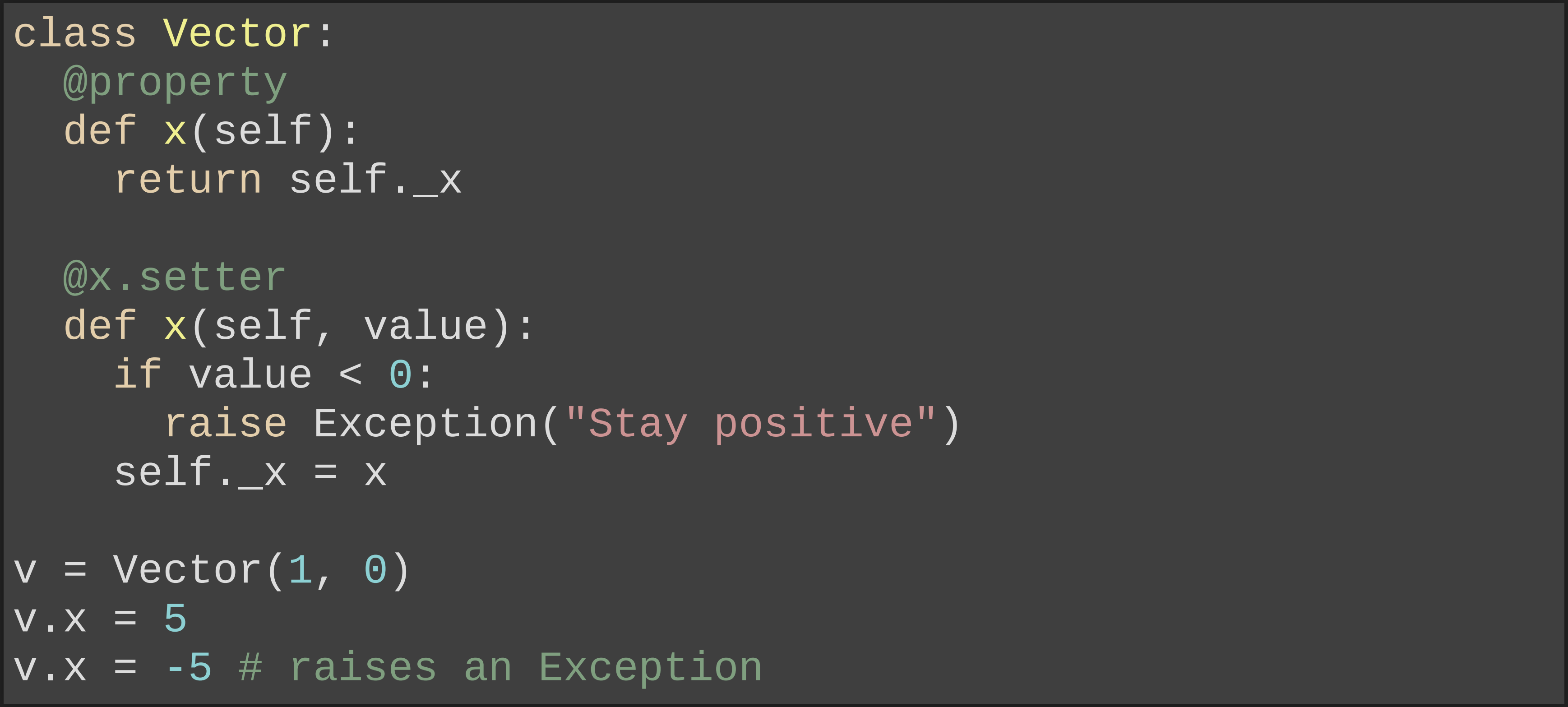
Properties

```
class Vector:
  @property
  def length(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)

v = Vector(1, 0)
x = v.length  # 1
```

```python
class Vector:
  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, value):
    if value < 0:
      raise Exception("Stay positive")
    self._x = x
```



```python
class Vector:
  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, value):
    if value < 0:
      raise Exception("Stay positive")
    self._x = x

v = Vector(1, 0)
v.x = 5
v.x = -5 # raises an Exception
```

Encapsulation (public/private)?

```
class Vector:
  # by convention private methods begin with _
  def _a(self):
    pass

  def __b(self):
    pass
```

Encapsulation (public/private)?

```
class Vector:
  # by convention private methods begin with _
  def _a(self):
    pass

  def __b(self):
    pass

v = Vector()
v._a()
# v.__b()           # doesn't work
v._Vector__b()      # mangled by interpreter
```

Polymorphism

```python
class Car:
  def get_wheels(self):
    return 4

class Motorcycle:
  def get_wheels(self):
    return 2
```

Polymorphism

```
class Car:
  def get_wheels(self):
    return 4

class Motorcycle:
  def get_wheels(self):
    return 2

def print_wheels(obj):
  print(obj.get_wheels())
```

Polymorphism

```
class Car:
  def get_wheels(self):
    return 4

class Motorcycle:
  def get_wheels(self):
    return 2

def print_wheels(obj):
  print(obj.get_wheels())

print_emissions(Car())
print_emissions(Motorcycle())
```

Inheritance

```
class DieselEngine:
  # constructor
  def __init__(self, fuel):
    # creating new attributes
    self.fuel = fuel

  def emissions(self):
    return 10

class VolksWagenEngine(DieselEngine):
  def emissions(self):
    # calling parent method
    return super().emissions() / 2
```

Multiple inheritance (💣)

```python
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")
```

Multiple inheritance (💣)

```
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")

class C(A, B): pass
```

Multiple inheritance (💣)

```
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")

class C(A, B): pass
class D(B, A): pass
```

Multiple inheritance (💣)

```python
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")

class C(A, B): pass
class D(B, A): pass

c = C()
```

Multiple inheritance (💣)

```python
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")

class C(A, B): pass
class D(B, A): pass

c = C() # A
```

Multiple inheritance (💣)

```
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")

class C(A, B): pass
class D(B, A): pass

c = C() # A
d = D()
```

Multiple inheritance (💣)

```
class A:
  def __init__(self):
      print("A")

class B:
  def __init__(self):
      print("B")

class C(A, B): pass
class D(B, A): pass

c = C() # A
d = D() # B
```

Inheritance is often harmful

Abstraction problems:

Inheritance is often harmful

Abstraction problems:

```
class Square: pass
class Rectangle(Square): pass
```

Inheritance is often harmful

Abstraction problems:

```python
class Square: pass
class Rectangle(Square): pass
```

or

Inheritance is often harmful

Abstraction problems:

```
class Square: pass
class Rectangle(Square): pass
```

or

```
class Rectangle: pass
class Square(Rectangle): pass
```

Inheritance is often harmful

Abstraction problems:

```
class Square: pass
class Rectangle(Square): pass
```

or

```
class Rectangle: pass
class Square(Rectangle): pass
```

?

Neither! Prefer composition:

```
class Square:
  def __init__(self, side):
    self.rect = Rectangle(side, side)


  def area(self):
    return self.rect.area()

  def set_side(self, side):
    self.rect.set_width(side)
    self.rect.set_height(side)
```

Neither! Prefer composition:

```python
class Square:
  def __init__(self, side):
    self.rect = Rectangle(side, side)

  # no need for interfaces, polymorphism is for free
  def area(self):
    return self.rect.area()

  def set_side(self, side):
    self.rect.set_width(side)
    self.rect.set_height(side)
```

Class blowup:

```
class File: pass
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
class GzippedUTF8EncodedFile(File): pass
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
class GzippedUTF8EncodedFile(File): pass
... madness!
```

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
class GzippedUTF8EncodedFile(File): pass
... madness!
```

+ code reuse

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
class GzippedUTF8EncodedFile(File): pass
... madness!
```

+ code reuse

– number of classes can grow quickly

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
class GzippedUTF8EncodedFile(File): pass
... madness!
```

+ code reuse

– number of classes can grow quickly

– cannot be changed at runtime easily

Class blowup:

```
class File: pass
class GzippedFile(File): pass
class UTF8EncodedFile(File): pass
class EncryptedFile(File): pass
class GzippedEncryptedFile(File): pass
class GzippedUTF8EncodedFile(File): pass
... madness!
```

+ code reuse

– number of classes can grow quickly

– cannot be changed at runtime easily

– when parent changes, you have to change

Solved by composition elegantly (Decorator pattern):

```python
class File: pass
```

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))
```

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
```

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass
```

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass

f = Encryptor(Gzipper(UTF8Encoder(File("out.txt"))))
```

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass

f = Encryptor(Gzipper(UTF8Encoder(File("out.txt"))))
f = Gzipper(File("out.txt"))
```

Solved by composition elegantly (Decorator pattern):

```python
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass

f = Encryptor(Gzipper(UTF8Encoder(File("out.txt"))))
f = Gzipper(File("out.txt"))
```

+ scales linearly

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass

f = Encryptor(Gzipper(UTF8Encoder(File("out.txt"))))
f = Gzipper(File("out.txt"))
```

- + scales linearly
- + easily changed at runtime

Solved by composition elegantly (Decorator pattern):

```
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass

f = Encryptor(Gzipper(UTF8Encoder(File("out.txt"))))
f = Gzipper(File("out.txt"))
```

- + scales linearly
- + easily changed at runtime
- + loose coupling

Solved by composition elegantly (Decorator pattern):

```python
class File: pass
class Gzipper:
  def __init__(self, file):
    self.file = file

  def write(self, data):
    self.file.write(gzip(data))

class UTF8Encoder: pass
class Encryptor: pass

f = Encryptor(Gzipper(UTF8Encoder(File("out.txt"))))
f = Gzipper(File("out.txt"))
```

+ scales linearly
+ easily changed at runtime
+ loose coupling
− code duplication (solvable with delegation)

Magic methods

```python
class Vector:
```

Magic methods

```python
class Vector:




print(Vector(1, 2))  # [1, 2]
```

Magic methods

```
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)










print(Vector(1, 2))  # [1, 2]
```

Magic methods

```
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)










print(Vector(1, 2))  # [1, 2]
print(len(Vector(1, 0)))  # 1
```

Magic methods

```
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)

  def __len__(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)







print(Vector(1, 2))  # [1, 2]
print(len(Vector(1, 0)))  # 1
```

Magic methods

```
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)

  def __len__(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)







print(Vector(1, 2))  # [1, 2]
print(len(Vector(1, 0)))  # 1
if Vector(1, 2):
  print("non-zero vector")
```

Magic methods

```python
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)

  def __len__(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)

  def __bool__(self):
    return self.x != 0 and self.y != 0





print(Vector(1, 2))  # [1, 2]
print(len(Vector(1, 0)))  # 1
if Vector(1, 2):
  print("non-zero vector")
```

Magic methods

```
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)

  def __len__(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)

  def __bool__(self):
    return self.x != 0 and self.y != 0




print(Vector(1, 2))  # [1, 2]
print(len(Vector(1, 0)))  # 1
if Vector(1, 2):
  print("non-zero vector")
print(Vector(1, 2) + Vector(3, 4))  # [4, 6]
```

Magic methods

```
class Vector:
  def __str__(self):
    return "[{}, {}]".format(self.x, self.y)

  def __len__(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)

  def __bool__(self):
    return self.x != 0 and self.y != 0

  def __add__(self, other):
    return Vector(self.x + other.x, self.y + other.y)

print(Vector(1, 2))  # [1, 2]
print(len(Vector(1, 0)))  # 1
if Vector(1, 2):
  print("non-zero vector")
print(Vector(1, 2) + Vector(3, 4))  # [4, 6]
```

Magic methods (context manager)

```
class DBTransaction:
```

Magic methods (context manager)

```python
class DBTransaction:





with DBTransaction() as tx:
  pass  # commit
```

Magic methods (context manager)

```
class DBTransaction:









with DBTransaction() as tx:
  pass  # commit

with DBTransaction() as tx:
  raise Exception()  # rollback
```

Magic methods (context manager)

```
class DBTransaction:
  def __enter__(self):
    self.begin()









with DBTransaction() as tx:
  pass  # commit

with DBTransaction() as tx:
  raise Exception()  # rollback
```

Magic methods (context manager)

```
class DBTransaction:
  def __enter__(self):
    self.begin()

  def __exit__(self, exc_type, exc_val, exc_tb):
    if exc_type is None:
      self.commit()
    else:
      self.rollback()

with DBTransaction() as tx:
  pass  # commit

with DBTransaction() as tx:
  raise Exception()  # rollback
```

Iterator protocol

```
for x in l:
  print(x)
```

Iterator protocol

```
for x in l:
  print(x)
```

```
# 'l' is 'iterable', 'it' is 'iterator'
it = iter(l)  # calls l.__iter__
while True:
  try:
    x = next(it) # calls l.__next__
    print(x)
  except StopIteration:
    break
```

Implementing iterators using generators

```
class ListIter:
  def __init__(self, list):
    self.list = list
```

Implementing iterators using generators

```
class ListIter:
  def __init__(self, list):
    self.list = list





for i in ListIter([1, 2, 3]):
  print(i)
```

Implementing iterators using generators

```
class ListIter:
  def __init__(self, list):
    self.list = list

  def __iter__(self):
    for x in self.list:
      yield x

for i in ListIter([1, 2, 3]):
  print(i)
```

Testing object type

```
isinstance(object, class)
isinstance(5, int)    # True
isinstance(6, str)    # False
```

Testing object type

```
isinstance(object, class)
isinstance(5, int)    # True
isinstance(6, str)    # False

class Base: pass
class Derived(Base): pass
```

Testing object type

```
isinstance(object, class)
isinstance(5, int)   # True
isinstance(6, str)   # False

class Base: pass
class Derived(Base): pass

isinstance(Base(), Base)
```

Testing object type

```
isinstance(object, class)
isinstance(5, int)   # True
isinstance(6, str)   # False

class Base: pass
class Derived(Base): pass

isinstance(Base(), Base)
isinstance(Derived(), Derived)
```

Testing object type

```python
isinstance(object, class)
isinstance(5, int)   # True
isinstance(6, str)   # False

class Base: pass
class Derived(Base): pass

isinstance(Base(), Base)
isinstance(Derived(), Derived)
isinstance(Base(), Derived)
```

Testing object type

```
isinstance(object, class)
isinstance(5, int)   # True
isinstance(6, str)   # False

class Base: pass
class Derived(Base): pass

isinstance(Base(), Base)
isinstance(Derived(), Derived)
isinstance(Base(), Derived)
isinstance(Derived(), Base)
```

Imports and package system 🤮

Imports and package system 🤮

```
import <module_name>
```

Imports and package system 🤮

```
import <module_name>
```

1. Directory where interpreter was launched

Imports and package system 🤮

```
import <module_name>
```

1. Directory where interpreter was launched
2. List of directories in env. var. PYTHONPATH

Imports and package system 🤮

```
import <module_name>
```

1. Directory where interpreter was launched
2. List of directories in env. var. PYTHONPATH
3. System paths

How to check?

How to check?

```python
import sys
print(sys.path)
```

How to check?

```
import sys
print(sys.path)
sys.append('/my/import/path')
```

How to check?

```
import sys
print(sys.path)
sys.append('/my/import/path')
import mylib # mylib is also searched in '/my/import/path'
```

How can you import?

How can you import?

```
import math
math.sqrt(5)
```

How can you import?

```python
import math
math.sqrt(5)
```

```python
from math import sqrt
sqrt(5)
```

How can you import?

```
import math
math.sqrt(5)
```

```
from math import sqrt
sqrt(5)

from math import sin as cos
```

How can you import?

```python
import math
math.sqrt(5)
```

```python
from math import sqrt
sqrt(5)

from math import sin as cos
```

```python
from math import *
sqrt(sin(5))
```

What happens when a module is imported?

What happens when a module is imported?

a.py

```
print("ahoj")
variable = 5
```

What happens when a module is imported?

a.py

```python
print("ahoj")
variable = 5
```

b.py

```python
import a # a.py is executed, 'ahoj' is printed
print(a.variable)  # 5
```

What happens when a module is imported?

a.py

```
print("ahoj")
variable = 5

if __name__ == "__main__":
  # someone executed python a.py directly
  print("hello from a.py")
```

b.py

```
import a # a.py is executed, 'ahoj' is printed
print(a.variable)  # 5
```

Directory organization (packages)

```
main.py
lib/
  __init__.py  # marks 'lib' as a package (< Python 3.3)
  sound.py
  graphics.py
```

Directory organization (packages)

```
main.py
lib/
  __init__.py  # marks 'lib' as a package (< Python 3.3)
  sound.py
  graphics.py
```

main.py

```python
import lib.sound
from lib.graphics import render
```

sound.py

```python
from .graphics import render # relative path must be used
```

Circular imports

Circular imports
chicken.py

```
from .egg import Egg

class Chicken:
  def gimme(self):
    return Egg()
```

Circular imports

chicken.py

```python
from .egg import Egg

class Chicken:
  def gimme(self):
    return Egg()
```

egg.py

```python
from .chicken import Chicken

class Egg:
  def hatch(self):
    return Chicken()
```

Circular imports

chicken.py

```python
from .egg import Egg

class Chicken:
  def gimme(self):
    return Egg()
```

egg.py

```python
class Egg:
  def hatch(self):
    from .chicken import Chicken  # local import
    return Chicken()
```

Python has a LOT of libraries built-in

Python has a LOT of libraries built-in

- Data structures

Python has a LOT of libraries built-in

- Data structures
- Synchronization (threads, ...)

Python has a LOT of libraries built-in

- Data structures
- Synchronization (threads, ...)
- Math

Python has a LOT of libraries built-in

- Data structures
- Synchronization (threads, ...)
- Math
- Filesystem

Python has a LOT of libraries built-in

- Data structures
- Synchronization (threads, ...)
- Math
- Filesystem
- Database (SQLite)

Python has a LOT of libraries built-in

- Data structures
- Synchronization (threads, ...)
- Math
- Filesystem
- Database (SQLite)
- CSV, XML, JSON

Python has a LOT of libraries built-in

- Data structures
- Synchronization (threads, ...)
- Math
- Filesystem
- Database (SQLite)
- CSV, XML, JSON
- Compression, cryptography, networking, HTTP, FTP, e-mail, GUI, tests, ...

Additional Python libraries can be found at PyPi (Python package index)

Additional Python libraries can be found at PyPi
(Python package index)
The easiest way to install is using **pip**

Additional Python libraries can be found at PyPi (Python package index)

The easiest way to install is using **pip**

```
$ pip install pytest
```

requirements.txt

```
requests
pygame
flask==1.0.2
```

requirements.txt

```
requests
pygame
flask==1.0.2
```

```
$ pip install -r requirements.txt
```

Python style 👗

PEP8 - universal standard

How to check?

How to check?

```
$ pip install flake8
$ flake8 f.py
```

How to check?

```
$ pip install flake8
$ flake8 f.py
f.py:2:1: F401 'math' imported but unused
f.py:8:1: W293 blank line contains whitespace
f.py:14:9: F841 local variable 'a' is assigned to but never us
f.py:16:1: W391 blank line at end of file
```

How to fix?

How to fix?

```
$ pip install autopep8
$ autopep8 f.py -i
```