UDACITY NANODEGREE

INTERMEDIATE PYTHON

# Python foundations

[Python foundations](#_rdowm0y7i4li)

[Intro](#_r97hbbys02p)

[Code style](#_zaayvb46eh8a)

[Refresher - Objects in Python](#_khp3wb4dj0cx)

[Classes in OOP](#_i3mkzc2u93w4)

[Unified Modeling Language](#_sjpc6lyny3by)

[Exceptions:](#_gunmqehnht0m)

[Docstrings](#_lcfjvzs2k77j)

## Intro

<https://www.youtube.com/watch?v=XSu73XIcfX0&t=108s>

Specifically, in this lesson, we will:

* Review some core elements of Pythonic design, including *object-oriented systems thinking*
* Learn how to write code in a consistent, scalable style, using the *PEP8* style guide and the *Pycodestyle* linter
* Learn how to create well-documented code using *Docstrings* and *Doctests*

The core Python language is defined by the [Python Language Reference](https://docs.python.org/3/reference/index.html#reference-index). These features can be used in any scripts with no need to *import* other modules. They include basic functionality like simple datatypes for primitives (such as strings and integers) as well as basic sequences (like lists and tuples).

[The Python Standard Library](https://docs.python.org/3/library/index.html#library-index) extends the functionality of the python language with many features that are commonly needed by programmers. These modules are "built into" Python, but they must be *imported* to the program when they're used. The [array](https://docs.python.org/3/library/array.html) module is one of many examples of a built-in standard library module.

Remember that compiled languages like C have a potential to be more efficient than Python. So using the standard library can often give you a performance boost, since much of it is actually implemented using C.

## Code style

Use linters: such as pip install pycodestyle

# Refresher - Objects in Python

**Object-Oriented Programming (OOP)** is a way for us to structure our code so that *data* and *behaviors* are related to one another in a logical (and usually more intuitive) way—allowing us to interact with our code as if it were composed of distinct *objects*. Many of the built-in and community-maintained libraries demonstrate object-oriented design, and for good reason: As our programs grow, making use of OOP design patterns can help ensure that our code remains modular, well-organized, and understandable.

### **Classes in OOP**

Classes are an object-oriented concept that groups data (**instance variables**) and actions that can be performed on that data (**instance methods**). The key idea is that the instance variables and methods are both related to a certain concept. We used the example of a cat, but we can have classes to represent all sorts of different concepts (e.g. documents, images, cars, random-number generators, and so on!). As our systems become larger, our classes can be combined and extended to produce more complex actions. However, it is important to consider the main **roles and responsibilities** of a class—aiming to keep each class small and easily maintained.

### **Unified Modeling Language**

As we're designing our software, we'll often attempt to represent the structure and relationships of data and classes visually. Often we'll use a visual design language known as [Unified Modeling Language (UML)](https://en.wikipedia.org/wiki/Unified_Modeling_Language) to achieve this goal. UML was developed in the 1990s to help developers express software concepts visually. UML has many use-cases including *class diagrams* to explain how classes interact, *sequence diagrams* to describe software execution order, and *use-case diagrams* to clarify how the software will be used. It is a common and useful tool that you'll likely encounter as you work on more complex projects.

Use the following code block to answer the question:

**class** **Document**():

**def** **\_\_init\_\_**(self, filename):

self.filename = filename

contract = Document('agreement3234.docx')

signed\_contract = contract

signed\_contract.filename = 'agreement3234\_signed.pdf'

print(contract.filename)

What will the output of the print function be?

* agreement3234.docx
* agreement3234\_signed.pdf
* <\_\_main\_\_.Document object at 0x10c3325c0>

### Exceptions:

import re

def validate\_email(email):

""" Test if an email is of a valid format """

if type(email) is not str:

raise Exception('Email is not a String')

regex = '^\w+([\.-]?\w+)\*@\w+([\.-]?\w+)\*(\.\w{2,3})+$'

if not re.search(regex,email):

raise Exception('Email is Malformed')

return True

>> Can also check for types of params, or just simply install mypy and will see the erroneous types by running mypy mymodule.py (same as black or flake8, shows errors, but also shows errors on hover without running it)

### Docstrings

**def** **divide**(a=1, b=2):

""" Returns the quotient of a divided by b

Arguments:

a {int} -- the numerator (defaults 1)

b {int} -- the denominator (defaults 2)

Raises:

Exception: if b is 0

"""

**if** b == 0:

**raise** Exception("Cannot divide by zero")

**return** a / b

First, we can install a command line utility to help us ensure that our docstrings are written properly: pip install pydocstyle

Then, we can run pydocstyle cat.py to list all errors in our docstrings.

### 

### Doctest (two versions - in docs vs in main.py)

Having a code block such as:

class Cat():

def \_\_init\_\_(self, name:str, age:int):

self.name = name

self.age = age

def speak(self) -> None:

print(f'{self.name} says, purrrrrr.')

To add doctests, we need to add some code to execute when running the script (to execute the test):

**if** \_\_name\_\_ **==** "\_\_main\_\_":

**import** doctest

doctest.testmod(extraglobs**=**{'kitty': Cat('Spot', 3)})

This code specifies that we want to run the included doctests, and it sets a global object that the tests can access at runtime.

Then we can extend the speak docstring to include a doctest that references the kitty global object:

def speak(self) -> None:

"""Make a cute cat sound.

>>> kitty.speak()

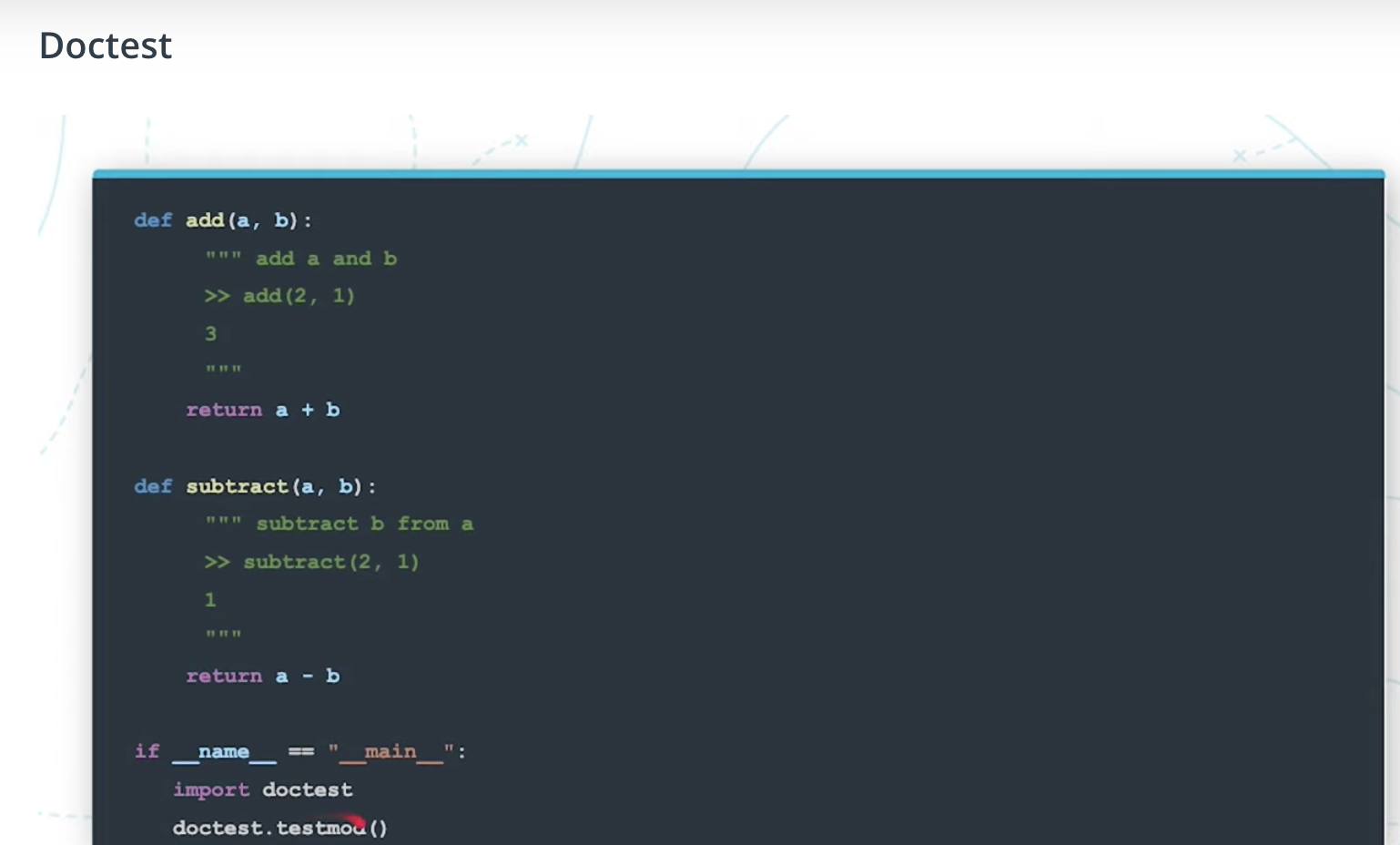
Spot says, purrrrrr

"""

print(f'{self.name} says, purrrrrr.')

If you run python3 cat.py you'll now see

OR (in docs directly):



### Modularising the code

When the import keyword is run by the Python interpreter, what is the first thing that happens *after the path is resolved*?

>> The corresponding \_\_init\_\_.py file is run

Python allows us to combine multiple import statements into one import statement:

**import** os, sys, random

However, this is considered bad practice ([PEP-8](https://python.org/dev/peps/pep-0008/#imports)) and it is preferred to import each module on it's own line:

**import** os

**import** sys

**import** random

Big code bases are better off in smaller modules of ~20-40 lines, so that we can see clearly what the functionalities and responsibilities of each of these smaller modules are. For that, we can not just create several xy.py files alone. We need to create an init.py file in the directory, even if we just keep it empty, but we need it there to be able to reference and import classes and functions from one file in the other one.

Example:

*Directory File Contents*:

\_\_init\_\_.py

main.py

aloha.py

adios.py

*Aloha.py*:

**def** say\_hi():

print('hello')

*Adios.py*:

**def** say\_goodbye():

print('byebye')

*Main.py*:

**from** aloha **import** say\_hi

**from** adios **import** say\_goodbye

say\_hi()

say\_goodbye()

Example 2:

Adding a nested directory greetings. We could import such as from greetings/aloha import say\_hi or we could include an init.py again in the nested directory too, and handle there all the relative imports (referencing them by a dot), and then we can run the main.py as following:

Directory File Contents:

\_\_init\_\_.py

main.py

greetings/

\_\_init\_\_.py

aloha.py

adios.py

*greetings/\_\_init\_\_.py:*

from .aloha import say\_hi

from .adios import say\_goodbye

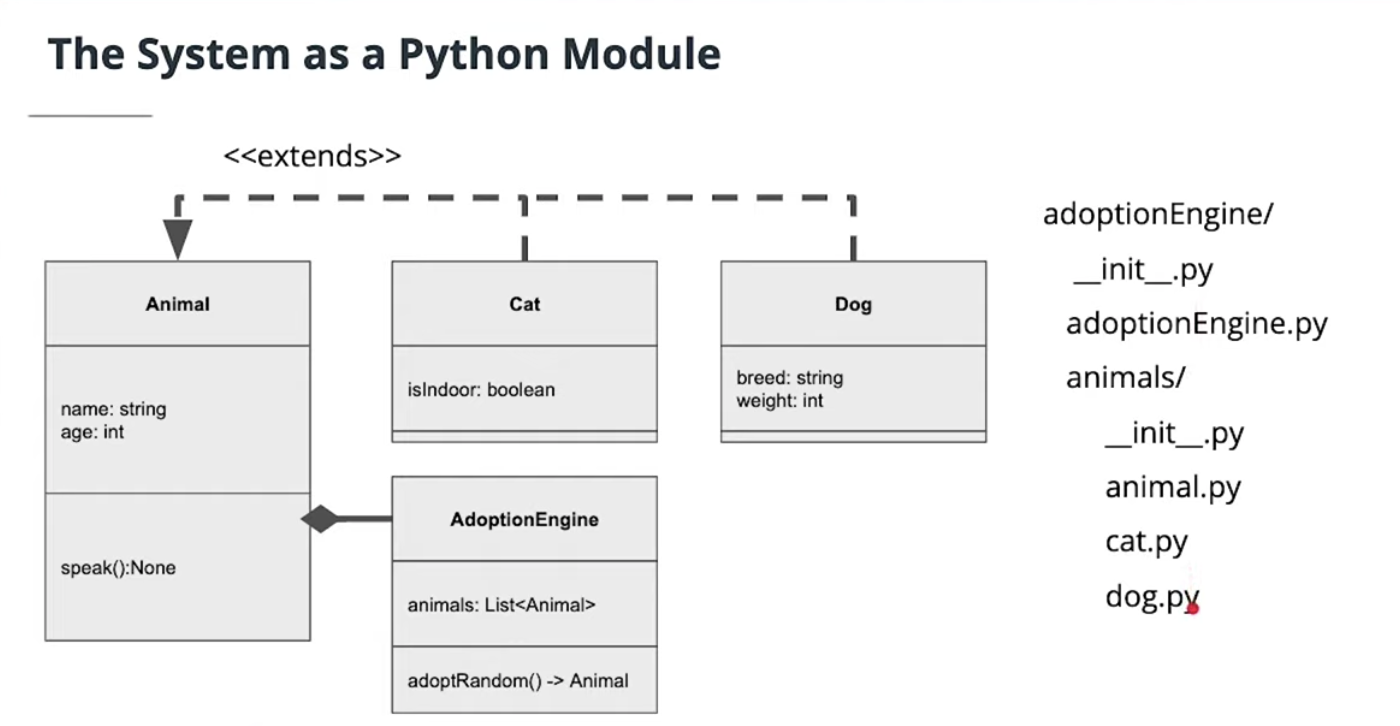
*main.py:*

from greetings import say\_hi, say\_goodbye

say\_hi()

say\_goodbye()

### System Thinking in Python : OOP Inheritance



Advanced OOP: Abstract classes:

<https://www.youtube.com/watch?time_continue=224&v=SSc0dP3Ufjo&feature=emb_logo>

In Python we can define an *abstract class* by inheriting the Python Standard Library's [Abstract Base Class (ABC)](https://docs.python.org/3/library/abc.html). This helps us out in two ways:

1. It prevents an abstract class from being instantiated.
2. It indicates that any abstract methods should be defined in the children classes.

To do this, we would first import ABC, and then we can have our Animal class inherit from it:

**from** abc **import** ABC

**class** **Animal**(ABC):

**pass**

Abstract classes allow us to define a clear, standard interface for similar types of classes, and ensure the methods are fully defined for child objects. More reading [here](https://www.geeksforgeeks.org/abstract-classes-in-python/)

Cropping image and adding text on top exercise:

# **Step 1: Study and Install the Library**

Now that we have our problem clearly defined. We'll need to find a library to help us. Our goal is to find a single library that is capable of:

1. Resizing and cropping an image
2. Adding text onto the image
3. Saving the resulting image as a jpg file

Take a moment to review the following libraries:

* [Numpy](https://numpy.org/doc/)
* [Pillow](https://pillow.readthedocs.io/en/stable/index.html)
* [Spacy](https://spacy.io/)

*Which library will best solve our problem?*

HIDE SOLUTION

The Solution is **Pillow**:

* [Numpy](https://numpy.org/doc/) offers powerful features for working with matricies. Images can be represented as a matrix of pixel values, so we can accomplish some simple tasks like resizing and cropping photos—but adding text will be a challenge.
* [**Pillow**](https://pillow.readthedocs.io/en/stable/index.html) **offers features to work with graphics and has interfaces for filtering, resizing, and drawing onto the images (including drawing text!)**
* [Spacy](https://spacy.io/) is a powerful tool for working with text and written words. We're interested in working with images, so this won't be very helpful.

*What is the command to install the selected library?*

HIDE SOLUTION

pip install Pillow

### **Try it!**

Before continuing, run the install command in the terminal window so the library is available as we continue to build!

# **Step 2: Using the Library to Resize Images**

After identifying a suitable library, let's break our development into a series of steps. Let's start with opening and performing some relatively simple transformation operations to crop and resize the input image.

Image

Using the Pillow documentation, see if you can find out how to use the library to resize and crop an image. Here are some links to point you in the right direction:

* [Image](https://pillow.readthedocs.io/en/stable/reference/Image.html#the-image-class) class
* [Image.open](https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.open) instance method
* [Image.size](https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.size) instance variable
* [Image.resize](https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.Image.resize) instance method
* [Image.crop](https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.Image.crop) instance method
* [Image.save](https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.Image.save) instance method

### **Try it!**

Your tasks are to complete the skeleton code (top right) to achieve the following requirements:

1. Load an image into a Pillow Image object.
2. Crop the image at ./imgs/img.jpg to center on the dog's face.
3. Resize the cropped image to be two times the size, while preserving the aspect ratio (the proportion of the image height to its width).
4. Save the image to a new jpg file.

HIDE SOLUTION

**from** PIL **import** Image

**def** generate\_postcard(in\_path, out\_path, crop**=None**, width**=None**):

"""Create a Postcard With a Text Greeting

Arguments:

in\_path {str} -- the file location for the input image.

out\_path {str} -- the desired location for the output image.

Returns:

str -- the file path to the output image.

"""

img **=** Image.open(in\_path)

**if** crop **is** **not** **None**:

img **=** img.crop(crop)

**if** width **is** **not** **None**:

ratio **=** width**/**float(img.size[0])

height **=** int(ratio**\***float(img.size[1]))

img **=** img.resize((width, height), Image.NEAREST)

img.save(out\_path)

**return** out\_path

**if** \_\_name\_\_**==**'\_\_main\_\_':

print(generate\_postcard('./imgs/img.jpg',

'./imgs/out.jpg',

(450, 900, 900, 1300),

200))

# **Step 3: Adding Text**

Let's complete our simple application and add additional complexity using more advanced features of Pillow to add a messsage onto our image. We can use Pillow to draw text and geometric shapes onto images.

Image

Once again, we should start by reviewing the documentation to understand how we can use the library for this goal. Here are some links to point you in the right direction:

* [ImageDraw](https://pillow.readthedocs.io/en/stable/reference/ImageDraw.html) module
* [ImageDraw.text](https://pillow.readthedocs.io/en/stable/reference/ImageDraw.html#PIL.ImageDraw.PIL.ImageDraw.ImageDraw.text) instance method
* [ImageFont](https://pillow.readthedocs.io/en/stable/reference/ImageFont.html#PIL.ImageFont.ImageFont) module
* [ImageFont.truetype](https://pillow.readthedocs.io/en/stable/reference/ImageFont.html#PIL.ImageFont.truetype) instance method

### **Try it!**

Your tasks are to continue to extend your code in img.py to:

1. Add a message onto the cropped image.
2. Add style to the message typography adjusting fill and size. We've included LilitaOne-Regular.ttf in the ./fonts directory. You can add additional fonts from [Google Fonts](https://fonts.google.com/).

HIDE SOLUTION

**from** PIL **import** Image, ImageDraw, ImageFont

**def** generate\_postcard(in\_path, out\_path, message**=None**, crop**=None**, width**=None**):

"""Create a Postcard With a Text Greeting

Arguments:

in\_path {str} -- the file location for the input image.

out\_path {str} -- the desired location for the output image.

crop {tuple} -- The crop rectangle, as a (left, upper, right, lower)-tuple. Default=None.

width {int} -- The pixel width value. Default=None.

Returns:

str -- the file path to the output image.

"""

img **=** Image.open(in\_path)

**if** crop **is** **not** **None**:

img **=** img.crop(crop)

**if** width **is** **not** **None**:

ratio **=** width**/**float(img.size[0])

height **=** int(ratio**\***float(img.size[1]))

img **=** img.resize((width, height), Image.NEAREST)

**if** message **is** **not** **None**:

draw **=** ImageDraw.Draw(img)

font **=** ImageFont.truetype('./fonts/LilitaOne-Regular.ttf', size**=**20)

draw.text((10, 30), message, font**=**font, fill**=**'white')

img.save(out\_path)

**return** out\_path

**if** \_\_name\_\_**==**'\_\_main\_\_':

print(generate\_postcard('./imgs/img.jpg',

'./imgs/out.jpg',

'woof!',

(450, 900, 900, 1300),

200))

# **Strategy Object Design Pattern**

## **Combining Our Skills - Encapsulating Third-Party Libraries**