CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 8

Thursday, September 30th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- The concept for consistency in the `python` language:
  - The `python` data model
  - Special class methods (also called *"dunder"* methods)
- A custom sequence: French deck of cards
- Software Licenses

OUTLINE

- Class methods, static methods and instance methods
- `python` modules
- `python` packages and the `python` package index (PyPI)

CLASS, STATIC AND INSTANCE METHODS

CLASS, STATIC AND INSTANCE METHODS

At this point you should feel comfortable with user-defined classes, special class methods and the `python` data model. You should develop an intuition for the *consistency* of the `python` programming language once you have understood these *key concepts*.

CLASS, STATIC AND INSTANCE METHODS

Recall the *decorator* design pattern discussed in Lecture 5:

- A decorator *wraps* hidden code around a function argument and returns a new decorated function object.
- `python` makes use of decorators to further specialize methods in a user-defined class.
- These decorators are available since `python 2.2` (new-style classes, see previous lecture).
- There are two such decorators (PEP318):
  - `@classmethod`: transforms a *method* into a *class method*.
  - `@staticmethod`: transforms a *method* into a *static method*.

CLASS, STATIC AND INSTANCE METHODS

Let us revisit the `Complex` class from Lecture 6:

```
class Complex:
    def __init__(self, real, imag):
        """Default initialization of a complex number"""
        self.real = real
        self.imag = imag

    @classmethod
    def make_complex(cls, real, imag):
        """Factory method for a complex number"""
        return cls(real, imag)

    def __repr__(self):
        """String representation"""
        return f"{type(self).__name__}({self.real}, {self.imag})"

    def __eq__(self, other):
        """Equality of two complex numbers"""
        return (self.real == other.real) and (self.imag == other.imag)
```

CLASS, STATIC AND INSTANCE METHODS

Let us revisit the `Complex` class from Lecture 6:

```python
class Complex:
    def __init__(self, real, imag):
        """Default initialization of a complex number"""
        self.real = real
        self.imag = imag

    @classmethod
    def make_complex(cls, real, imag):
        """Factory method for a complex number"""
        return cls(real, imag)
```

- `make_complex` is a special *class method*; a result from the `@classmethod` decorator.
- Note the difference in the *function signature*:
  - Regular methods take `self` as first argument (***reference to instance of class***).
  - Class methods take `cls` as first argument (***reference to class type***).
- The name `cls` is again *a convention* just as `self` is chosen by convention.
- The return value `cls(real, imag)` is the same as `Complex(real, imag)`. The `@classmethod` decorator *strips away* the reference to the *instance of the class* and returns a reference to the `Complex` class type instead.

CLASS, STATIC AND INSTANCE METHODS

Example usage of our decorated class method:

```
>>> z1 = Complex(1, 2) # calls __init__
>>> z2 = Complex.make_complex(1, 2) # create an instance from the Complex type directly
>>> z3 = z2.make_complex(1, 2) # create an instance via another instance
>>> z1 == z2 and z2 == z3 # the three instances are all equal
True
```

The instances are all equal *but* they are separate objects in memory:

```
>>> id(z1); id(z2); id(z3)
140022962355792
140022964138048
140022964039344
```

CLASS, STATIC AND INSTANCE METHODS

Takeaway:

- A `@classmethod` has access to *what is defined in the class itself*, but *no access to the **state** of a particular instance of the class*. (Note that `self` is a reference to ***state***, this state is initialized only when `__init__` is called and may change over time. The `cls` reference *does not* have access to such state.)
- The main use case for `@classmethod` is to provide *alternate* ways of constructing an object of your class:
  - The original client only needed the Cartesian form to create complex numbers.
  - A new client requires complex numbers to be constructed from Polar coordinates (radius r and angle φ). This new class feature is implemented with a `@classmethod`.

CLASS, STATIC AND INSTANCE METHODS

Construct complex numbers from Cartesian and Polar coordinates:

```python
import numpy as np

class Complex:
    def __init__(self, real, imag):
        """Default initialization of a complex number"""
        self.real = real
        self.imag = imag

    @classmethod
    def from_polar(cls, r, phi):
        """Construct a complex number from Polar coordinates"""
        return cls(r * np.cos(phi), r * np.sin(phi))

    def __repr__(self):
        """String representation"""
        return f"{type(self).__name__}({self.real}, {self.imag})"

    def __eq__(self, other):
        """Equality of two complex numbers"""
        return (self.real == other.real) and (self.imag == other.imag)
```

CLASS, STATIC AND INSTANCE METHODS

Construct complex numbers from Cartesian and Polar coordinates:

```python
import numpy as np

class Complex:
    def __init__(self, real, imag):
        """Default initialization of a complex number"""
        self.real = real
        self.imag = imag

    @classmethod
    def from_polar(cls, r, phi):
        """Construct a complex number from Polar coordinates"""
        return cls(r * np.cos(phi), r * np.sin(phi))
```

```python
>>> z1 = Complex(np.cos(np.pi / 4), np.sin(np.pi / 4))
>>> z2 = Complex.from_polar(1, np.pi / 4)
>>> z1 == z2
True
```

CLASS, STATIC AND INSTANCE METHODS

The `@staticmethod` decorator is similar but *strips away* the first argument completely:

```
class MyClass:
    def __init__(self):
        """Default initialization of MyClass with reference to an instance"""
        print(self)

    @classmethod
    def class_method(cls):
        """Class method with a reference to MyClass"""
        print(cls)

    @staticmethod
    def static_method(): # no first argument here!
        """Static methods are just normal functions in the scope of the class"""
        pass
```

CLASS, STATIC AND INSTANCE METHODS

```python
class MyClass:
    def __init__(self):
        """Default initialization of MyClass with reference to an instance"""
        print(self)

    @classmethod
    def class_method(cls):
        """Class method with a reference to MyClass"""
        print(cls)

    @staticmethod
    def static_method(): # no first argument here!
        """Static methods are just normal functions in the scope of the class"""
        pass
```

```python
>>> c = MyClass()
<__main__.MyClass object at 0x7f06d0ce1b20>
>>> MyClass.class_method()
<class '__main__.MyClass'>
>>> type(MyClass.static_method)
<class 'function'>
```

CLASS, STATIC AND INSTANCE METHODS

```
>>> c = MyClass()
<__main__.MyClass object at 0x7f06d0ce1b20>
>>> MyClass.class_method()
<class '__main__.MyClass'>
>>> type(MyClass.static_method)
<class 'function'>
```

- Static methods are just *normal functions* inside the class scope (`MyClass` in this example).
- You can call them directly from the class type like `MyClass.static_method()` ***or*** from an instance like this `c.static_method()`.
- Static methods in `python` are the same as `C++` methods declared with the `static` keyword.
- Can you use static methods to modify state?

CLASS, STATIC AND INSTANCE METHODS

Let's look at it from another perspective (*Fluent Python*):

```python
class Demo:
    def instance_method(*args):
        return args

    @classmethod
    def class_method(*args):
        return args

    @staticmethod
    def static_method(*args):
        return args
```

```python
>>> d = Demo()
>>> dummy_args = ('A', 'B', 'C')
>>> d.instance_method(*dummy_args)
(<__main__.Demo object at 0x7fec186bab80>, 'A', 'B', 'C') # fist arg: reference self
>>> d.class_method(*dummy_args)
(<class '__main__.Demo'>, 'A', 'B', 'C') # first arg: reference to cls
>>> d.static_method(*dummy_args)
('A', 'B', 'C') # static methods: no implicit first argument like other methods do!
```

CLASS VARIABLES AND INSTANCE VARIABLES

- Static methods are used for global class operations that *do not depend on state* (an instance of the class carries state).
- Just as there are *static* and *instance* methods for a class, there are also ***static variables (class variables)*** and ***instance variables***.
- Class variables are *global* to the class itself (just like static methods), whereas instance variables are *local to the class instance* (they represent state and may hold different values for different instances of the class).

CLASS VARIABLES AND INSTANCE VARIABLES

Recall the French deck class from the previous lecture:

```python
from collections import namedtuple

Card = namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    """French deck of 52 playing cards"""
    ranks = [str(rank) for rank in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        """Initialize ordered deck of cards"""
        self._cards = [
            Card(rank, suit) for suit in self.suits for rank in self.ranks
        ]

    # skipping other methods shown in previous lecture
```

CLASS VARIABLES AND INSTANCE VARIABLES

Recall the French deck class from the previous lecture:

```python
from collections import namedtuple

Card = namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    """French deck of 52 playing cards"""
    # the following are class variables, there is no `self.` in front!
    ranks = [str(rank) for rank in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        """Initialize ordered deck of cards"""
        # this is an instance variable, remember: `self` is a reference to an instance
        self._cards = [
            Card(rank, suit) for suit in self.suits for rank in self.ranks
        ]
```

Note that *class variables (global to the class type itself)* do not have a `self.` prepended, because `self` is a *reference to an instance of the class*.

- `ranks` and `suits` are *global (class) properties*. The card deck will always consist of 52 cards.
- `self._cards` *is a state* that is local to the instance because the deck might be shuffled differently between two instances, hence their *state* is different.

CLASS VARIABLES AND INSTANCE VARIABLES

Class variable and instance variable example:

```
>>> class Demo:
...     class_variable = 1 # a class variable (global to the type Demo)
>>> demo = Demo() # create an instance of Demo
>>> demo.class_variable = 2 # this shadows Demo.class_variable
>>> demo.class_variable # the value 2 is now local to the instance!
2
>>> demo.__class__.class_variable # but the global class variable is still untouched
1
```

CLASS VARIABLES AND INSTANCE VARIABLES

Class variable and instance variable example:

```
1  >>> class Demo:
2  ...     class_variable = 1 # a class variable (global to the type Demo)
3  >>> demo = Demo() # create an instance of Demo
4  >>> demo.class_variable = 2 # this shadows Demo.class_variable (duck-typing)
5  >>> demo.class_variable # the value 2 is now local to the instance!
6  2
7  >>> demo.__class__.class_variable # but the global class variable is still untouched
8  1
9  >>> demo.__class__.class_variable = 3 # change the class variable globally
10 >>> # Note that this is the same statement: Demo.class_variable = 3
11 >>> new_demo = Demo()
12 >>> new_demo.class_variable
13 3
```

Same example on pythontutor

Note: the reason the code in `line 4` ***shadows*** the class variable is because of ***duck typing*** in `python`. The duck typing rules create a new `self.class_variable` attribute *for the instance*, you can see that in the pythontutor example above.

CLASS VARIABLES AND INSTANCE VARIABLES

Class variable and instance variable example:

- We can further investigate this duck typing phenomenon with the `dir()` and `vars()` built-in functions.
- `dir()`: lists the names of the class attributes and recursively of its base classes:

```
>>> dir(demo)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'class_variable']
>>> dir(new_demo)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'class_variable']
```

- `vars()`: lists the contents of the `__dict__` attribute (local to instance):

```
>>> vars(demo)  # affected by the duck-typing phenomenon (see line 4 in previous slide)
{'class_variable': 2}
>>> vars(new_demo) # we did not duck-type anything on this instance
{}
```

CLASS, STATIC AND INSTANCE METHODS

Summary:

- `@classmethod` is primarily used as a *factory* to create new class instances in different ways other than how you define it in `__init__()`. In the `@classmethod` you perform the desired transformation first and then create a new instance by calling `__init__` with the result of your data transformation.
- `@classmethod` does not need to return an instance of `cls` all the time, it is just often used this way.
- ***Optional reading:*** The factory pattern is further described in Chapter 3 of *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison Wesley Professional, 1995.
- `@staticmethod` are regular functions that are contained within the class scope. You can either call them via an instance `self.static_method()` (assuming `self` is an instance of `MyClass`) or via the class type directly `MyClass.static_method()`.

python MODULES

`python` MODULES

- We are now at a point where we can take our `python` knowledge one step further.
- You learned about the basic `python` language features such as defining functions, writing user-defined types (classes) all aligned with the consistency enabled by the `python` data model.
- This knowledge allows you develop large software projects already but when you are starting to scale up your code, ***structure is important***.
- Just like when you are cooking for a large party, you want your mise en place in the kitchen (`python` modules and packages) and sharp knifes (your dev environment, tools, editor or IDE).
- `python` *modules* contain subsets of your code project.
- `python` *packages* are a collection of modules. This collection is often *hierarchical* in the same way as your Linux filesystem is and they can form components of your software projects.

python MODULES

- A `python` module most of the time is a simple `python` file with code inside, e.g. `my_module.py`. You could execute a module with `python my_module.py`.
- The more common use case is to `import` a module in your code where you need the *functionality* provided by that module:

```
import my_module
retval = my_module.some_function() # use a function implemented in my_module
```

Note: `some_function` is *inside* the namespace of `my_module`.

- You could have done this to *import into the current namespace* (***but you know it is bad practice!***):

```
from my_module import *
retval = some_function() # use a function imported from my_module
```

python MODULES

Good practice:

- Import only the functionality you actually need:

```
from my_module import some_function
retval = some_function() # use a function imported from my_module
```

Importing into the current namespace does not prevent from ***name clashes***

- It is usually a better idea to *keep the namespace* of the module. We are *lazy* typists—use the `as` keyword to make your life easier:

```
import my_module as mm
retval = mm.some_function() # use a function implemented in my_module
```

- You may have seen this many times with more widely used packages: (these are again *conventions* that the `python` community sticks with)

```
import numpy as np # numerical python package (linear algebra, regression, etc.)
import pandas as pd # data analysis package
import matplotlib.pyplot as plt # powerful plotting package
```

- Other useful `python` packages: `scipy` (scientific library), `sympy` (symbolic math), `numba` (performance)

python MODULES

Where does python look for modules:

- `python` searches some system dependent locations for modules:

```
>>> import sys
>>> print(sys.path)
['', '/usr/lib/python39.zip', '/usr/lib/python3.9', '/usr/lib/python3.9/lib-dynload',
'/home/fabs/.local/lib/python3.9/site-packages', '/usr/lib/python3.9/site-packages']
```

  - `''` : current directory
  - `/home/fabs/.local/lib/python3.9/site-packages` : version dependent user directory for packages on Linux. Everything you install with `python -m pip install --user` goes there. To find out the user base of your `python` installation run `python -m site --user-base` .
  - The others are *system* directories. Anything you install via your *package manager* or by `sudo python -m pip install` goes there.
- Use the `PYTHONPATH` environment variable to extend the search path to your own locations. This is also very useful for *development*: point the `PYTHONPATH` to the module/package you are developing in order to skip installing it into the default search path all the time! `PYTHONPATH` behaves the same as `PATH` , you know how that works.
- If a module can not be found, you will get a `ModuleNotFoundError` .

python MODULES

When and in what order should you `import` modules:

- If you need to import other modules in your module, you should import *after* your module's documentation.
- The order of imported modules should be as follows:
  1. Standard library modules
  2. Third-party modules: `numpy`, `pandas`, `pytorch`, etc.
  3. Your own modules

python MODULES

Example: simple module in current directory – `module_1.py`

```python
"""
Docstring for module_1
"""

import numpy as np

# the __all__ attribute will be honored when somebody executes
#      from module_1 import *
# it will import only what you specify in the __all__ list
__all__ = ['foo']

pi = np.pi # \pi in module scope

def foo():
    print(f"module_1.foo(): pi = {pi:.6f}")

def bar():
    print(f"module_1.bar(): pi = {pi:.6f}")
```

python MODULES

Example: simple module in current directory – `module_1.py`

```python
import numpy as np

__all__ = ['foo']

pi = np.pi # \pi in module scope

def foo():
    print(f"module_1.foo(): pi = {pi:f}")

def bar():
    print(f"module_1.bar(): pi = {pi:f}")
```

```python
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
>>> from module_1 import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'foo']
>>> foo()
module_1.foo(): pi = 3.141593
```

python PACKAGES

`python` **PACKAGES**

- Modules are a great way to organize your code into *logical* units.
- This one-level organization is usually not deep enough for larger projects.
- `python` packages allow you to organize your project *hierarchically*, just like you would organize your code in your project directory on your file system (this is not a coincidence).

python PACKAGES

Example package hierarchy:

```
cs107_package
├── __init__.py
├── subpkg_1 # sub-package
│     ├── __init__.py
│     ├── module_1.py
│     └── module_2.py
└── subpkg_2 # sub-package
      ├── __init__.py
      ├── module_3.py
      ├── module_4.py
      └── module_5.py
```

python PACKAGES

Example package hierarchy:

```
cs107_package
├── __init__.py
├── subpkg_1 # sub-package
│     ├── __init__.py
│     ├── module_1.py
│     └── module_2.py
└── subpkg_2 # sub-package
      ├── __init__.py
      ├── module_3.py
      ├── module_4.py
      └── module_5.py
```

- The root of the `python` package
- You could also have modules on this level (there are none in this example).

python PACKAGES

Example package hierarchy:

```
cs107_package
├── __init__.py
├── subpkg_1 # sub-package
│     ├── __init__.py
│     ├── module_1.py
│     └── module_2.py
└── subpkg_2 # sub-package
      ├── __init__.py
      ├── module_3.py
      ├── module_4.py
      └── module_5.py
```

- A sub-package within your package. It again contains a number of modules.
- You could have another sub-packages inside here.

python PACKAGES

The `__init__.py` file:

- The `__init__.py` is used for package-level initialization either when your package is imported *or* a module within the package is imported.
- You write normal `python` code in that file which is then executed when the package is imported (once).
- You can use the `__all__` list inside `__init__.py` to define what should be imported when someone does `from cs107_package import *`.
- Often the file is empty. Since `python 3.3` you do not need to have the file ***if*** it is empty.

python PACKAGES

How to import nested packages:

- You import a package or a nested sub-package just like a module:

```
>>> import cs107_package.subpkg_1.module_1
>>> cs107_package.subpkg_1.module_1.foo() # call a function inside that module, phew!
```

- ***This can be tedious!*** Use the `__init__.py` file to make the life your customer enjoyable.

python PACKAGES

How to import nested packages:

Let's assume we have this code in our modules:

- cs107_package/subpkg_1/module_1.py:

```python
# of course you can have classes in your modules
class Foo:
    pass

def foo():
    print("cs107_package.subpkg_1.module_1.foo()")
```

- cs107_package/subpkg_1/module_2.py:

```python
# note the '..': relative import in packages
from ..subpkg_2 import module_3 as mod3

def bar():
    mod3.baz()
    print("cs107_package.subpkg_1.module_2.bar()")
```

- cs107_package/subpkg_2/module_3.py:

```python
def baz():
    print("cs107_package.subpkg_2.module_3.baz()")
```

You could then write your `__init__.py` files like this:

- cs107_package/__init__.py:

```python
# note the '.': relative import in packages
from .subpkg_1 import (foo, bar)
from .subpkg_2 import baz

__all__ = ['foo', 'bar', 'baz']
```

- cs107_package/subpkg_1/__init__.py:

```python
from .module_1 import foo
from .module_2 import bar

__all__ = ['foo', 'bar']
```

- cs107_package/subpkg_2/__init__.py:

```python
from .module_3 import baz

__all__ = ['baz']
```

python PACKAGES

How to import nested packages:

With this structure defined in out `__init__.py` files, we can use our package in a more natural way we are used to from other packages we work with:

```
>>> import cs107_package as pkg
>>> pkg.bar()
cs107_package.subpkg_2.module_3.baz()
cs107_package.subpkg_1.module_2.bar()
```

Compare to:

```
>>> import numpy as np
>>> dir(np)
# a lot of output...
```

We can use the `__init__.py` files to define what we want to export from our code and what should remain hidden in the package. The top-level `__init__.py` of `numpy 1.21.2` contains 429 lines of code.

python PACKAGES

- Let us enter the `python` interpreter, and investigate the `__name__` attribute:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> __name__
'__main__'
>>> import module_1 as mod1
>>> mod1.__name__
'module_1'
```

  - The top-level environment in `python` is called `'__main__'`.
  - When we ***import*** a module, its name is set to the module filename without suffix.
- When we pass the module through the `python` interpreter directly, the module `__name__` attribute will be set to `'__main__'` by the interpreter. The difference to the example above is that *we do not import the module into another namespace*. It allows us to add functionality to modules when run by the interpreter. This is why you may have seen code like this at the end of a module:

```
if __name__ == "__main__":
    main() # function to be run when module is passed to the interpreter
```

python PACKAGES

- We can do the same with *packages*
- Because a package is a hierarchy of files and directories, we must implement this functionality in the `__main__.py` *file.*
- This file is then passed to the interpreter whenever we pass the `-m` option.

  Assume this is our `__main__.py` file in our test package:

```
import datetime as dt
print(f"Hello from cs107_package! Today is: {dt.datetime.now()}")
```

  We can then do this:

```
$ python -m cs107_package
Hello from cs107_package! Today is: 2021-09-21 19:45:15.069913
```

  This is exactly what happens when you use the `pip` package, for example.

```
$ python -m pip # runs the __main__.py file in the pip package
```

More reading on `python` modules and packages can be found in the `python` tutorial.

INSTALL AND DISTRIBUTE python PACKAGES

INSTALL AND DISTRIBUTE `python` PACKAGES

- You now know what `python` modules and packages are for.
- You are still missing the tools to properly *install* and *distribute* packages.
- Most `python` packages are available through the Python Package Index (PyPI). It is simply a remote server to fetch the software from.
- By default, `python -m pip install <package>` obtains the package from PyPI.
- Because PyPI is a ***production platform*** you should use the PyPI testing server when playing around with `pip`. Use it like this:
  - Simple package without dependencies:

```
$ python -m pip install --index-url https://test.pypi.org/simple/ your-package
```

  - If you need to resolve dependencies:

```
$ python -m pip install --index-url https://test.pypi.org/simple/ \
    --extra-index-url https://pypi.org/simple/ your-package
```

INSTALL AND DISTRIBUTE `python` PACKAGES

There are many ways in `python` to create packages and distribute them. The main documentation you should consult is:

- Installing packages: https://packaging.python.org/tutorials/installing-packages/
- Packaging projects: https://packaging.python.org/tutorials/packaging-projects/

The main tool to install packages in `python` is `pip`. The classical way is through `distutils/setuptools` but `pip` can handle these cases as well and should be preferred. There are two parts to `pip`:

1. `pip` itself is a ***frontend*** for installing `python` packages.
2. It uses a ***backend*** to accomplish this task.

The backend is *modular*, it can be `setuptools`, for example, or anything else that conforms to PEP517.

INSTALL AND DISTRIBUTE python PACKAGES

The basics steps to create a release that is publishable on PyPI:

1. Add a `pyproject.toml` file to your project (PEP518)
2. Install `build`: `python -m pip install build` (a PEP517 package builder)
3. Build your next package release: `python -m build .`
4. Upload to PyPI: `twine upload dist/*` (use https://test.pypi.org/)

Steps 1 and 2 need to be done only once. To create a new release, this is sufficient (steps 3 and 4):

```
$ rm dist/* && python -m build && twine upload dist/*
```

(`twine` can be installed via `pip`)

The `dist/` directory contains the built distributions. There are two distinctions:

1. Source distributions: contains source code only
2. Binary distributions: called ***wheels***

INSTALL AND DISTRIBUTE `python` PACKAGES

Assume we have this project structure: file:

```
python_project
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
└── src
    └── cs107_package
        ├── __init__.py
        ├── __main__.py
        ├── subpkg_1
        │   ├── __init__.py
        │   ├── module_1.py
        │   └── module_2.py
        └── subpkg_2
            ├── __init__.py
            ├── module_3.py
            ├── module_4.py
            └── module_5.py
```

The `cs107_package` is our `python` package from the previous discussion.

INSTALL AND DISTRIBUTE `python` PACKAGES

Create a `pyproject.toml` file:

- This file is used to specify the minimum build system requirements
- It is defined in PEP518
- For our example the `pyproject.toml` file looks like this:

```
[build-system]
requires = [
    "setuptools>=42",
    "wheel"
]
build-backend = "setuptools.build_meta" # use setuptools for building
```

INSTALL AND DISTRIBUTE python PACKAGES

For `setuptools` we need to create a setup.cfg file:

```
[metadata]
name = cs107_package
version = 0.0.2
author = Fabian Wermelinger
author_email = fabianw@seas.harvard.edu
description = A small example package
long_description = file: README.md
long_description_content_type = text/markdown
url = https://harvard-iacs.github.io/2021-CS107/
classifiers =
    Intended Audience :: Developers
    Programming Language :: Python :: 3
    Topic :: Software Development :: Libraries :: Python Modules

[options]
package_dir =
    = src
packages = find:
python_requires = >=3.6

[options.packages.find]
where = src
```

See https://packaging.python.org/tutorials/packaging-projects/#configuring-metadata

INSTALL AND DISTRIBUTE `python` PACKAGES

Building and distributing the project is now easy:

```
$ python -m build # omitting output
$ ls -1 dist/
cs107_package-0.0.2-py3-none-any.whl
cs107_package-0.0.2.tar.gz
$ twine upload --repository testpypi dist/*
```

- The package is now available at https://test.pypi.org/project/cs107-package/
- Note that we published on the *testing server*
- Once you have published a release (version) you can not overwrite it if you found a mistake. You must create a new release for this.
- You can install the package with:

```
$ python -m pip install -i https://test.pypi.org/simple/ cs107-package
```

RECAP

- Class methods, static methods and instance methods
- `python` modules
- `python` packages and the `python` package index (PyPI)