SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 9

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, September 29th 2022

LAST TIME

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

TODAY

Main topics: ***Classes and member functions (methods) revisited, Python modules and packages, Python Package Index***

Details:

- Revisiting classes:
  - Class methods
  - Static class methods
  - Instance methods
- Python modules
- Python packages and the Python Package Index (PyPI)

AGENDA CHECK:

- Milestone M1B is due on Tuesday 10/04 → first touch with GitHub Actions to prepare your project repository for task automation using continuous integration (CI).

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 6:

- 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) useful in *classes*:
  - `@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 7:

```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)

    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 7:

```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* (`cls`) 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:** `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. ***Example in practice:***
  - 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 would be implemented with a new constructor using `@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*:

```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
```

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
```

```
>>> 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 the difference 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') # bound method → self
>>> d.class_method(*dummy_args)
(<class '__main__.Demo'>, 'A', 'B', 'C') # class method → cls
>>> d.static_method(*dummy_args)
('A', 'B', 'C') # static methods: no explicit first argument!
```

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
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
        ]
```

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 → duck typing!
>>> 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 Python tutor 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 our duck-typing on 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 `@classmethod` you perform the desired transformation first and then create a new instance by calling `__init__` with the result of your data transformation.
- `@classmethod` may 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

- 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***.
- For your projects, you should structure the code into modules which can be imported. Many modules are then packed together into a *package*. All larger Python projects (NumPy, Matplotlib, Pandas, etc.) are packages.
- 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 used this form to *import into the current namespace* (***you should generally try to avoid this form, 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:

```python
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. To minimize typing entropy → use the as keyword to make your life easier:

```python
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)

```python
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`. *Make sure you also add the `bin` directory of your user base to `PATH`.*
  - 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` in the shell!
- If a module can not be found, you will get a `ModuleNotFoundError`.

PYTHON MODULES

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

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

PYTHON MODULES

Example: simple module in current directory, call it `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, call it `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

- Modules are a great way to organize your code into *logical* units.
- The one-level organization of modules is typically 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.

PYTHON PACKAGES

Example package hierarchy:

```
cs107_package # main package
├── __init__.py
├── subpkg_1 # sub-package
│   ├── __init__.py
│   ├── module_1.py # simple module file
│   └── 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 # main package
├── __init__.py
├── subpkg_1 # sub-package
│   ├── __init__.py
│   ├── module_1.py # simple module file
│   └── 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 # main package
├── __init__.py
├── subpkg_1 # sub-package
│   ├── __init__.py
│   ├── module_1.py # simple module file
│   └── 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-package inside here as well.

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 executes `from cs107_package import *`.
- Often the file is empty. Since Python 3.3 you do not need to have the `__init__.py` file ***if*** it is empty.

PYTHON PACKAGES

How to import nested packages:

- Packages (or nested sub-packages) are imported the same as modules:

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

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

- ***This can be tedious!*** Use the `__init__.py` file to make the life for your users a bit easier or use the `as` keyword to define an alias for code where the `__all__` list is not specified.

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']
```

- Often it makes sense *to keep the names local to the sub-package* → e.g. `np.linalg`

PYTHON PACKAGES

How to import nested packages:

- With this structure defined in our `__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
>>> dir(pkg)
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__', 'bar', 'baz', 'foo',
'subpkg_1', 'subpkg_2']
```

- Compare to NumPy:

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

- Use the `__init__.py` files to define what you want to export from our code and what should remain hidden in the package. For example, 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 *execute* the module with the Python interpreter directly, the module `__name__` attribute will be set to `'__main__'` by the interpreter. When the module is executed *it is not imported into another namespace* and hence its name is set to the string `'__main__'`. This allows you to add code to your modules that is run only when the module is executed by the interpreter directly. It 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

- The same can be done with Python *packages*
- Because a package is a hierarchy of (module) files and directories, you must implement this functionality in the `__main__.py` *file*.
- This file is then *executed by the interpreter* whenever you pass it the `-m` option on the command line. Assume the `__main__.py` in our test package contains:

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

We can execute the package `__main__.py` like 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
```

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. *(Up to now we have made things work using `PYTHONPATH` only.)*
- 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 (https://pypi.org/).
- Because PyPI is a ***production platform*** you should use the test server (https://test.pypi.org/) 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 from the PyPI server add `--extra-index-url`:

```
$ 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 old-school style of installing packages is (was) via `distutils/setuptools`. `pip` can work with these old methods as well and you should therefore prefer `pip`. *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 (e.g. `setuptools`).

- 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 required to create a release of a Python package 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 consecutive new release, this is sufficient (steps 3 and 4):

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

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

Aside: the `dist/` directory contains the built distributions that will be uploaded to the server. 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:

```
python_project
├── LICENSE
├── pyproject.toml  # project build and meta data
├── README.md
├── setup.cfg  # build configuration for setuptools backend
└── src
    └── cs107_package  # the actual Python code for the 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. *(You can find this package in supplementary code for this lecture in the class repository.)*

INSTALL AND DISTRIBUTE PYTHON PACKAGES

Create a `pyproject.toml` file:

- This file is used to specify the *minimum build system requirements*. It can also contain project meta data such as version, authors, etc.
- It is defined in PEP518
- For our simple test project 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 = 1.0.0
author = Fabian Wermelinger
author_email = fabianw@seas.harvard.edu
description = CS107/AC207 test package
long_description = file: README.md
long_description_content_type = text/markdown
url = https://code.harvard.edu/CS107/main/tree/master/lecture/code/lecture09/python_project
classifiers =
    Intended Audience :: Developers
    Programming Language :: Python :: 3
    Topic :: Software Development :: Libraries :: Python Modules

[options]
package_dir =               ; for our test package we chose to put the source code into `src`
    =src
packages = find:            ; required to automatically find our package in `src`
install_requires =          ; additional package dependencies (examples)
    python_version>="3.6"  ; minimum Python version requirement (example)
    numpy                  ; package requires numpy (example)

[options.packages.find]    ; additional config required for find above
where = src
```

INSTALL AND DISTRIBUTE PYTHON PACKAGES

Building and distributing the project is now easy:

```
$ python -m build # omitting output
$ ls -1 dist/
cs107_package-1.0.0-py3-none-any.whl
cs107_package-1.0.0.tar.gz
$ twine upload --repository testpypi dist/*
```

- After `twine upload` → the new package version is available at https://test.pypi.org/project/cs107-package/ *(note that used the testing server for publishing this example, not the production 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 this command:

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

The `-i` option tells `pip` which *index* to use. It must be the test index here because our package is only available there. You would install *production projects* without this option.

RECAP

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

Further reading:

- Chapter 9 in Luciano Ramalho, *"Fluent Python: Clear, Concise, and Effective Programming"*, O'Reilly Media, 2015
- Python modules and packages (***recommended***): https://docs.python.org/3/tutorial/modules.html#modules
- Distributing Python modules: https://docs.python.org/3/distributing/index.html#distributing-python-modules
- Configuring `setuptools` using `setup.cfg`: https://setuptools.pypa.io/en/latest/userguide/declarative_config.html
- Configuring meta data using `pyproject.toml`: https://packaging.python.org/tutorials/packaging-projects/#configuring-metadata