Documentation, modules, & packages in Python

Data Sciences Institute
University of Toronto

Simeon Wong

Course objective

How to write robust software in a team that we, our colleagues, and the public can trust and use with confidence.

Course overview

- [x] 1. Version control with Git
- [x] 2. Command-line Python & Configuration
- [x] 3. Raising errors
- [x] 4. Testing software
- [x] 5. Application programming interfaces
- [ ] **6. Documentation, modules, and packages**

Documentation in Python

Why documentation?

"Code is more often read than written."
— Guido van Rossum

- Code is read by yourself (in the future) and by other developers who use it or contribute to it
- Code that cannot be properly understood ...
  - e.g. its required inputs, its methods, and its outputs
- ... can be used in situations or in ways that result in errors or mistakes

Commenting vs Documenting code

- **Comments** describe your code for other developers
- It makes it easier to understand how (algorithmically) and why
- Helps with understanding intention, purpose, and design rationale

- **Documentation** describes your code for your users
- What your code does (for a user) and how to use it

Both are important!

Documentation

Python docstrings

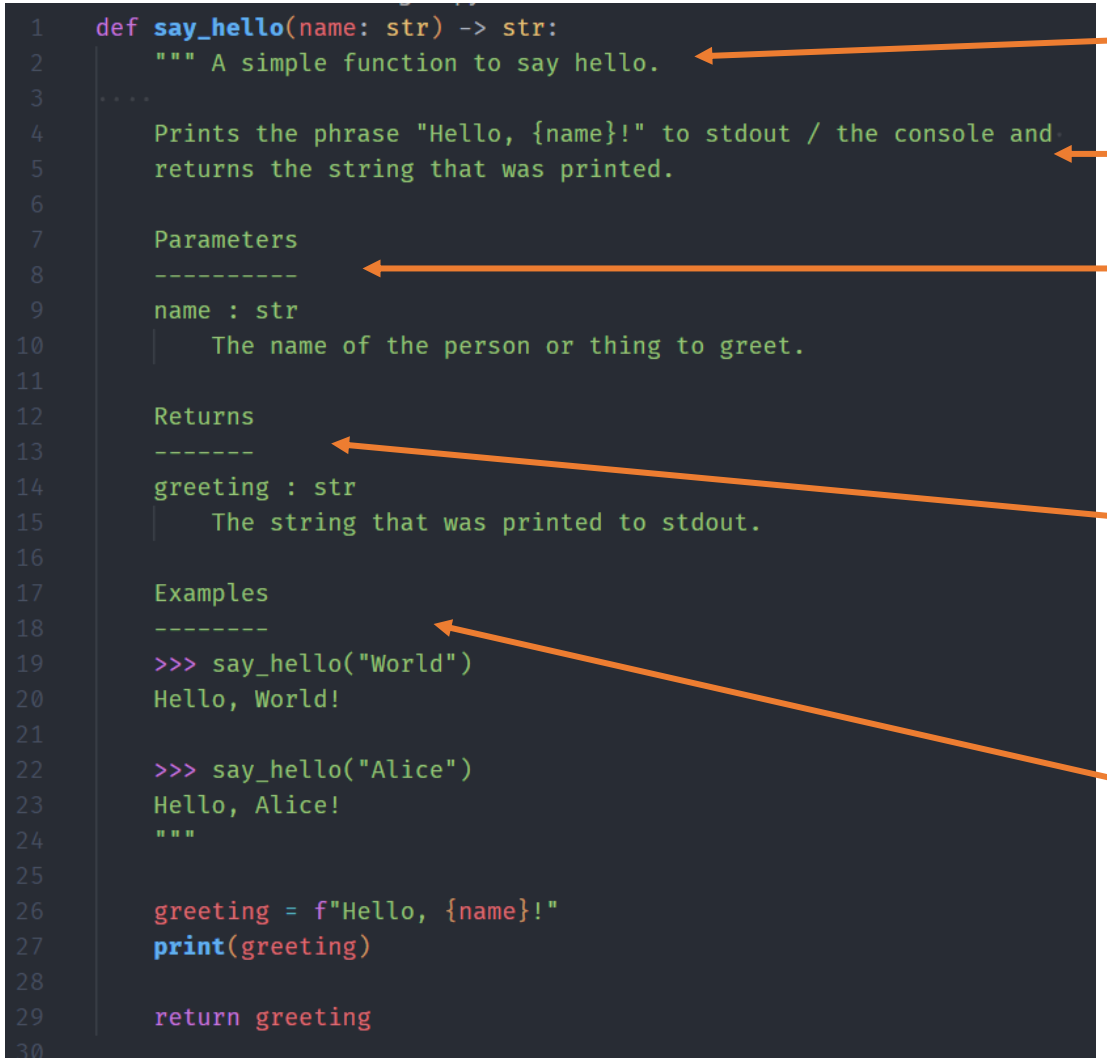
- Official Python standard to describe your code's functionality
- Wrapped in '''triple apostrophes or quotation marks''' at the very beginning of a function

```
def say_hello():
    ''' A simple function to say hello to the world '''
    print("Hello, World!")
```

Reference: https://peps.python.org/pep-0257/

Documentation

The numpy docstring style



```python
def say_hello(name: str) -> str:
    """ A simple function to say hello.

    Prints the phrase "Hello, {name}!" to stdout / the console and
    returns the string that was printed.

    Parameters
    ----------
    name : str
        The name of the person or thing to greet.

    Returns
    -------
    greeting : str
        The string that was printed to stdout.

    Examples
    --------
    >>> say_hello("World")
    Hello, World!

    >>> say_hello("Alice")
    Hello, Alice!
    """

    greeting = f"Hello, {name}!"
    print(greeting)

    return greeting
```

Short summary (max one-line)

Extended summary / description of function

List of parameters formatted as:

```
param_name : param_type
        Description of parameter
```

List of returned values formatted as:

```
value_type
    Description of this value
```

Optional examples

Reference: https://numpydoc.readthedocs.io/en/latest/format.html

Documentation

Python type hints

- The Python standard for documenting the types of values expected by your function
- Concise way of representing a subset of the information from the numpy docstring

`name` should be a `str`

Function returns a `str`

```
def say_hello(name: str) -> str:
```

References: https://peps.python.org/pep-0484/

Documentation

$> Interactive live coding

- Refactor the Hello World program (from BRS1) as a function
- Rewrite the command-line arguments as function parameters
  - Use Python type hints
- Write a docstring for the function
- Call the Hello World function based on the parsed arguments

Documentation
$> Original

```
import argparse

parser = argparse.ArgumentParser(description='Say hello to someone.')
parser.add_argument('name',
                    default='World',
                    type=str,
                    nargs='?',
                    help='Name to greet')
parser.add_argument('--repeat',
                    '-r',
                    type=int,
                    default=1,
                    help='Number of times to greet')
parser.add_argument('--goodbye',
                    '-g',
                    action='store_true',
                    help='Say goodbye instead of hello')

args = parser.parse_args()

message = 'Goodbye' if args.goodbye else 'Hello'

for _ in range(args.repeat):
    print(f'{message} {args.name}!')
```

Documentation

$> Refactored

```python
import argparse

parser = argparse.ArgumentParser(description='Say hello to someone.')
parser.add_argument('name',
                    default='World',
                    type=str,
                    nargs='?',
                    help='Name to greet')
parser.add_argument('--repeat',
                    '-r',
                    type=int,
                    default=1,
                    help='Number of times to greet')
parser.add_argument('--goodbye',
                    '-g',
                    action='store_true',
                    help='Say goodbye instead of hello')

args = parser.parse_args()

def print_greeting(name: str, repeat: int, goodbye: bool) -> None:
    ''' Print a greeting to the console.

    Parameters
    ----------
    name : str
        The name of the person to greet.
    repeat : int
        The number of times to greet the person.
    goodbye : bool
        If True, say goodbye instead of hello.

    Returns
    -------
    None
    '''
    message = 'Goodbye' if goodbye else 'Hello'

    for _ in range(repeat):
        print(f'{message} {name}!')

print_greeting(args.name, args.repeat, args.goodbye)
```

Sphinx documentation generator

- The most common web-based / online documentation generator for Python projects
- Parses your docstrings and other indicated text files
- Compiles into HTML files
  - Easier to read and browse
  - Easier to share online (or on an internal site) with your users
- Requires a moderate amount of setup to get started

Modules revisited

Instances of classes

- Also known as Object-Oriented Programming
- Keeps information about one logical item together
  - A single MRI acquisition
  - An employee in an HR application
  - A machine learning model + current state

Python modules
Why objects?

Without

```
person1_name = "John"
person1_role = "teacher"
person1_id = 198384
person1_supervisor_id = 100312
person1_salary = 50000
person1_location = "C16"

person2_name = "Jane"
person2_role = "teacher"
person2_id = 298384
person2_supervisor_id = 100312
person2_salary = 60000
person2_location = "F05"

def generate_tax_statement(name, role, id, supervisor_id, salary,
location):
    tax = salary * 0.18
    return f'''
    Employee: {name}
    Role: {role}
    ID: {id}
    Supervisor ID: {supervisor_id}
    Salary: {salary}
    Location: {location}
    Tax: {tax}
    '''

print(generate_tax_statement(person1_name, person1_role, person1_id,
person1_supervisor_id, person1_salary, person1_location))
```

Need to pass / work with 6 variables

With objects

```
def Employee():
    name: str
    role: str
    id: int
    supervisor_id: int
    salary: int
    location: str

    def __init__(self, name, role, id, supervisor_id, salary, location):
        self.name = name
        self.role = role
        self.id = id
        self.supervisor_id = supervisor_id
        self.salary = salary
        self.location = location

    def generate_tax_statement(self):
        tax = self.salary * 0.18
        return f'''
        Employee: {self.name}
        Role: {self.role}
        ID: {self.id}
        Supervisor ID: {self.supervisor_id}
        Salary: {self.salary}
        Location: {self.location}
        Tax: {tax}
        '''

person1 = Employee("John", "teacher", 198384, 100312, 50000, "C16")
print(person1.generate_tax_statement())

person2 = Employee("Jane", "teacher", 298384, 100312, 60000, "F05")
print(person2.generate_tax_statement())
```

Break / Exercise

Option 1: Refactor your homework into a module

- What data might be convenient to keep together in an object?
- What are some methods that could operate on that data?
- Write a docstring for your functions in the numpy style

Option 2: Try Sphinx

```
git clone https://github.com/UofT-DSI/building_software
cd "building_software/lessons/5 - Python documentation and packages/demos/sphinx-example"
sphinx-build
```

Option 3: Snack break

Python packages

Python packages

A simple Python package

- A generic package folder hierarchy
  - **Usual repo files**: LICENSE and README.md
  - **Package metadata**: pyproject.toml
  - **Code**: mycode.py
- You code can contain functions, static values, classes, etc...

```
packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
└── mycode.py
```

Reference: https://packaging.python.org/en/latest/tutorials/packaging-projects/

A larger Python package

- A generic package folder hierarchy
  - **Usual repo files**: LICENSE and README.md
  - **Package metadata**: pyproject.toml
  - **Source code**: src/
    - Directory with multiple source code files
  - **Automated tests**: tests/

```
packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── src/
│   └── example_package/
│       ├── __init__.py
│       └── example.py
└── tests/
```

Reference: https://packaging.python.org/en/latest/tutorials/packaging-projects/

Python packages

pyproject.toml

Template is copied from Python documentation

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "mymodule"
authors = [
    {name = "Simeon Wong",
     email = "simeon.wong@mail.utoronto.ca"},
]
description = "A very basic Python package"
version = "0.1.0"
dependencies = ["matplotlib", "numpy"]
requires-python = ">=3.10.0"
readme = "README.md"
classifiers = [
    "License :: OSI Approved :: MIT License",
]

[project.scripts]
myhelloworld = "mymodule:hello_world"
```

Which installer to use. Python recommendation is **hatchling**.

Project metadata that we fill in

Dependencies (packages we use in our code)

License info

Should our code be available as a bash command?

Reference: https://packaging.python.org/en/latest/tutorials/packaging-projects/

Installing/testing our package

- Install the package
  - in a fresh Python installation (to make sure it's compatible with everyone)
  - in editable mode (for easy debugging)

```
pip install -e /path/to/package
```

- After installation, the code can be imported

```
from my_package import my_module
```

$> Interactive live coding

- Download the **simple_package** code from the Building Software repo
- Personalize `pyproject.toml` file with your details
- Install it using `pip install -e ./`
- Try running `hello_world` from the command line

Bonus exercise:

- Add in `argparse` from your version of hello world you wrote during the configuration lesson

Distributing packages

- Python has a default repository: the Python Package Index (PyPI)
  - Packages are publicly and freely available to everyone
  - When you run `pip install`, it usually downloads from there
  - Beware of installing packages indiscriminately.
    - Anybody can contribute a package to PyPI
    - Although eventually delisted, there have been malicious packages
- Companies and research groups will often have private repositories, or keep packages in a private GitHub Organization for internal use

Semantic Versioning

- Semantic versioning is a widely adopted notation for indicating versions or changes to software
- Standard format: major.minor.patch (e.g. version 1.2.1)
  - **Major**: Incompatible API changes (breaking changes)
  - **Minor**: New functionality, where the existing usage is not affected
  - **Patch**: Bug-fixes and other changes with little impact to the user
- **Predictable upgrades and backwards compatibility**
  - *"Will everything break if I upgrade?"*

$> Interactive live coding

- Create a simple Python package from our Hello World code
- Install using `pip`

Extended exercise:

- Upload to GitHub, then install using `pip` directly from GitHub
  - Uninstall first, or use a new conda environment

Logistics

Course final assignment

- Available on GitHub
  https://github.com/UofT-DSI/building_software/blob/main/assignments/Assignment.md
- The final assignment is an extension of the homework so far
- Due at the end of this course:
  Sunday, Feb 18th at 23:59:59 EST (before midnight)

Course overview

- [x] 1. Version control with Git
- [x] 2. Command-line Python & Configuration
- [x] 3. Raising errors
- [x] 4. Testing software
- [x] 5. Application programming interfaces
- [ ] **6. Documentation, modules, and packages**

Course objective

How to write robust software in a team that we, our colleagues, and the public can trust and use with confidence.

Go forth and build great software! 💪

References

- Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson (https://merely-useful.tech/py-rse/config.html)