Testing software, and building Python packages

Data Sciences Institute
University of Toronto

Simeon Wong

UNIVERSITY OF TORONTO

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

☑ 1. Version control with Git

☑ 2. Command-line Python & Configuration

☑ 3. Raising errors

☑ 4. Testing software

☑ 5. Application programming interfaces

6. Modules and packages

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

Documentation

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

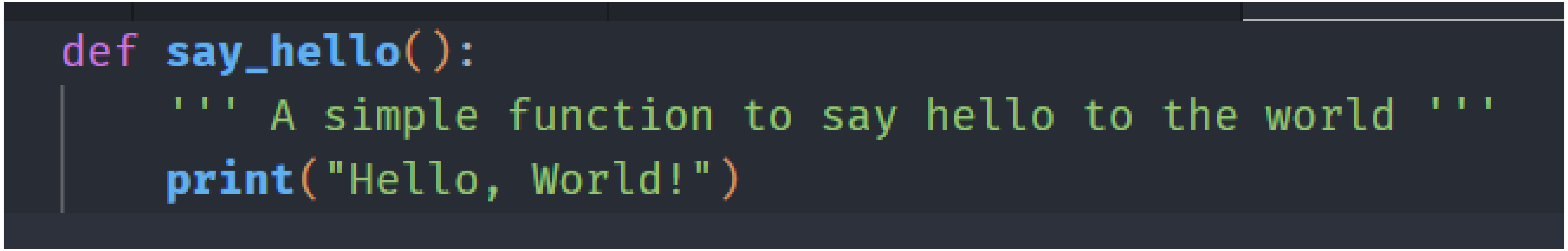
- **Documentation** describes your code for your users
- What your code does (for a user) and how to use it

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

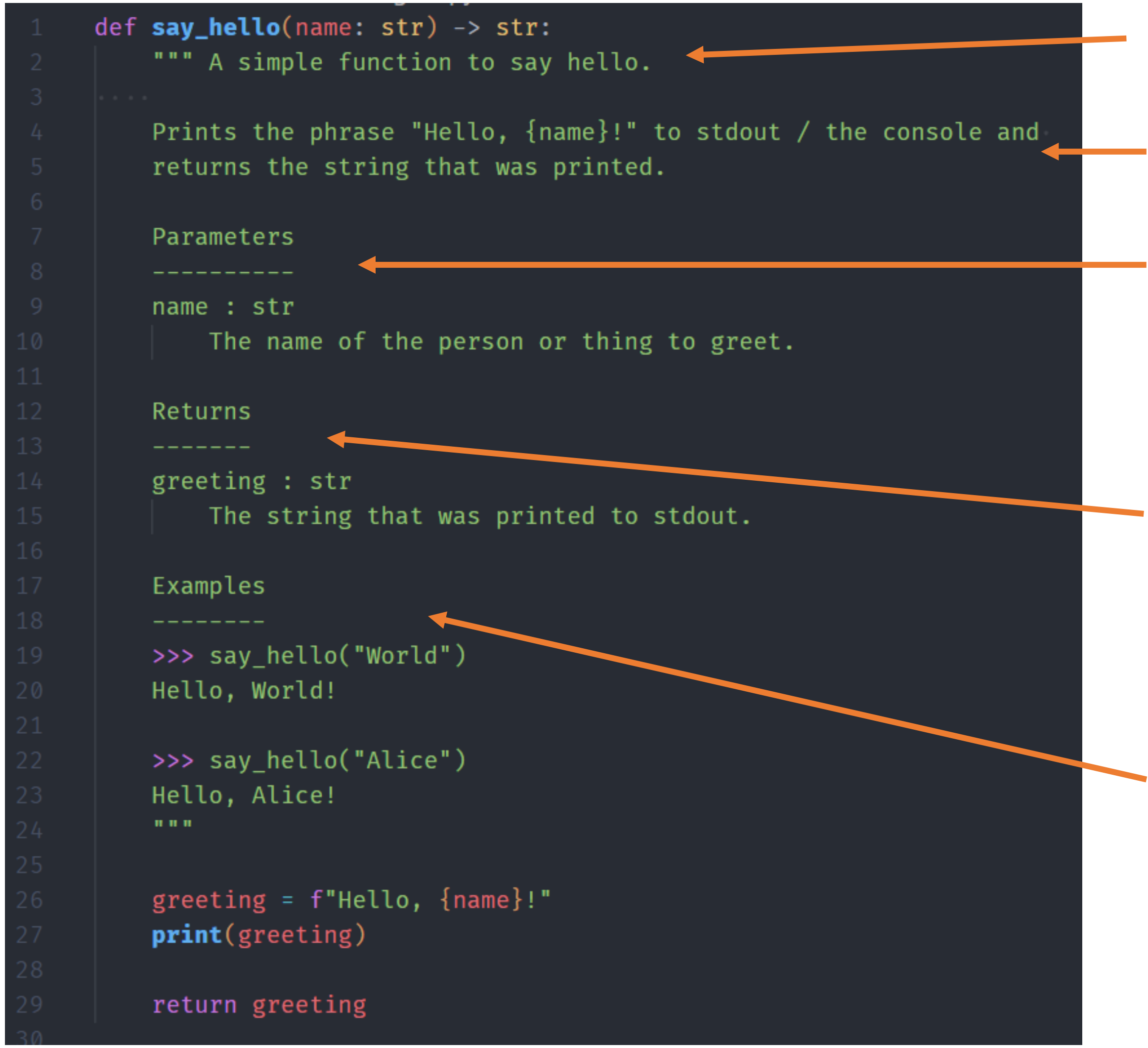


```python
def say_hello():
    ''' A simple function to say hello to the world '''
    print("Hello, World!")
```

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

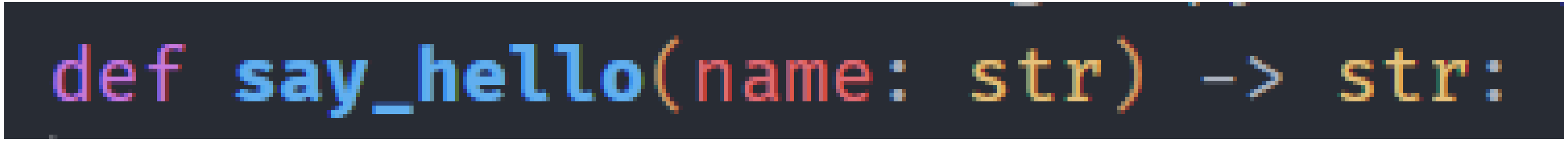
Documentation

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

```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()

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)
```

Documentation

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

__init__.py

Directory structure

Importing

Documentation

Break / Exercise

Option 1: Refactor your homework into a module

- Which functions and parameters are needed?
- 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
```

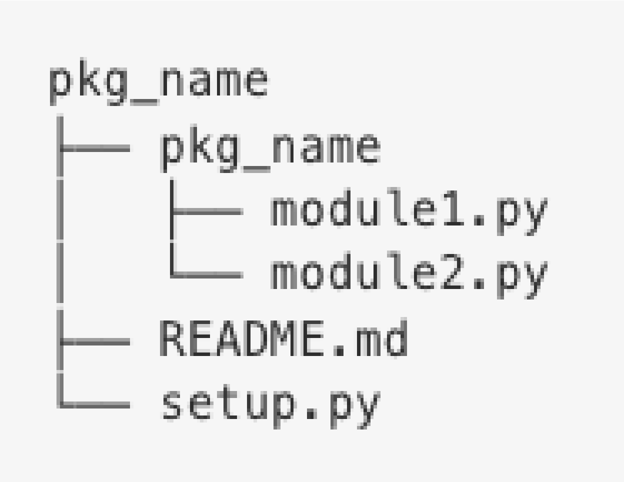
Option 3: Snack break

Python packages

Python packages

Creating a Python package

- A generic package folder hierarchy has:
  - a top-level directory with the package name
  - ↳ that contains a directory that is also named after the package
  - ↳ that contains the package's source files
- Modules can contain functions and classes

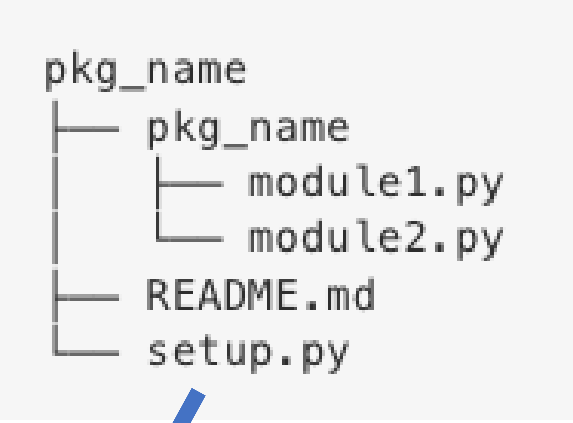


```
pkg_name
├── pkg_name
│   ├── module1.py
│   └── module2.py
├── README.md
└── setup.py
```

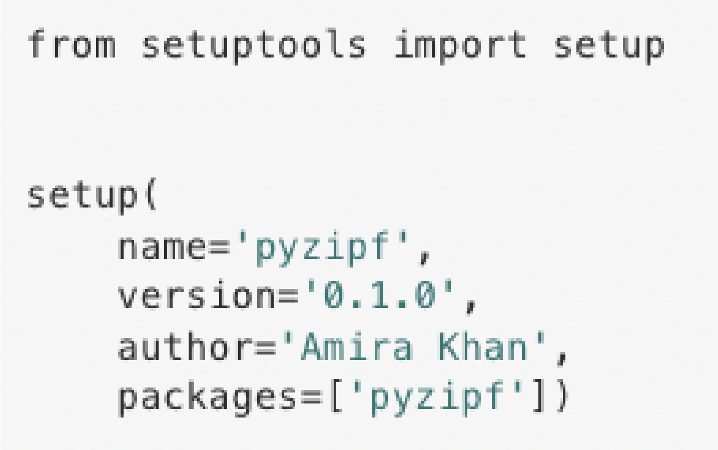
Python packages

setuptools

- Using setuptools allows everyone, regardless of Python distribution, to use our package
- setuptools attempts to install package requirements automatically
- The setup.py file defines some properties of our package



```
pkg_name
├── pkg_name
│   ├── module1.py
│   └── module2.py
├── README.md
└── setup.py
```



```
from setuptools import setup


setup(
    name='pyzipf',
    version='0.1.0',
    author='Amira Khan',
    packages=['pyzipf'])
```

Testing packages

- 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
```

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

Python packages
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
- Install using `pip`
- Upload to GitHub
- Install using `pip` in another instance

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 Sunday, Feb 18th at 23:59:59 EST (before midnight)

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)