Programming Course and Project

Summer Term 2024/25

Tutorial 4 - Documentation & Debugging

Felix Lundt - May 12, 2025

Tentative outline for the first phase

	Content Software Carpentry	Algorithm/ Game Play	General
Week 1 April 14	Project Setup		Intro, Python & Numpy primer
Week 2 April 28	TDD	Code skeleton	
Week 3 May 5	Git `game_utils.py` example	Code to play Random Agent Algorithms	
Week 4 May 12	Debugging & Documentation		Exam Registration (May 12th)
Week 5 May 19	Profiling		Submission Prototype (end of week)

Plan for today

- Exam registration
- Documentation
- Debugging
- Info on submission
- Status check

Documenting/commenting code

- Why?
  "Code is more often read than written." — *Guido van Rossum*
  Documentation is meant to help yourself and others use your code.

- Documenting vs. commenting:
  - Comments are for developers, explain code and its purpose and design.
    "Code tells you how; comments tell you why." — *Jeff Atwood*
  - Documentation explains the functionality to users.

- Take a function as an example:
  - Docstring explains what the function is doing and how it is used (arguments, return values, raised exceptions etc.)
  - Comments provide details about implementation

- Most important to document code: Good structure, good names and type hints!

Comments

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:  # True if i is 0 or a power of 2.
```

Block comment:
- starts with a # and space
- complete sentences
- two spaces after period
- paragraphs separated by #

Inline comment:
- two spaces from code
- start with # and space
- complete sentence(s)
- two spaces after period

Rules by Jeff Atwood:

- Comments close to code they refer to
- No complex formatting
  (No tables etc.)
- No redundant information
  (Assume reader understands the language, but not what the code is trying to do)
- Code should comment itself as much as possible
  (by proper naming, logical order, following conventions etc.)

The best comments are the ones you don't need ;)

```
# BAD COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1
```

```
x = x + 1                     # Increment x
```

```
x = x + 1                     # Compensate for border
```

Comments

```python
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:  # True if i is 0 or a power of 2.
```

Block comment:
- starts with a # and space
- complete sentences
- two spaces after period
- paragraphs separated by #

Inline comment:
- two spaces from code
- start with # and space
- complete sentence(s)
- two spaces after period

Rules by Jeff Atwood:

- Comments close to code they refer to
- No complex formatting
  (No tables etc.)
- No redundant information
  (Assume reader understands the language, but not what the code is trying to do)
- Code should comment itself as much as possible
  (by proper naming, logical order, following conventions etc.)

```
~~# BAD COMMENT: Now go through the b array and make sure whenever i occurs~~
~~# the next element is i+1~~
```

```
x = x + 1                   # ~~Increment x~~
```

```
x = x + 1                   # Compensate for border
```

The best comments are the ones you don't need ;)

Docstrings

- Conventions in PEP 257
- Docstrings are stored in `__doc__` variable
- Are formatted within triple double quotation marks (""" """)
- Purpose: provide overview and explain usage
- Tradeoff:
  Long enough to be helpful, but concise and short enough to be readable. Code should be designed to make this easy (good names are crucial!).
- Are annoying to maintain -> should reflect status of project
- Categories:
  - class docstrings
  - package and module docstrings
  - script docstrings
- Different standard styles (I recommend NumPy/SciPy)
  - NumPy/SciPy
  - Google
  - reStructured Text
  - Epytext
- In this course: class and function docstrings

Docstrings

Brief summary

Class attributes

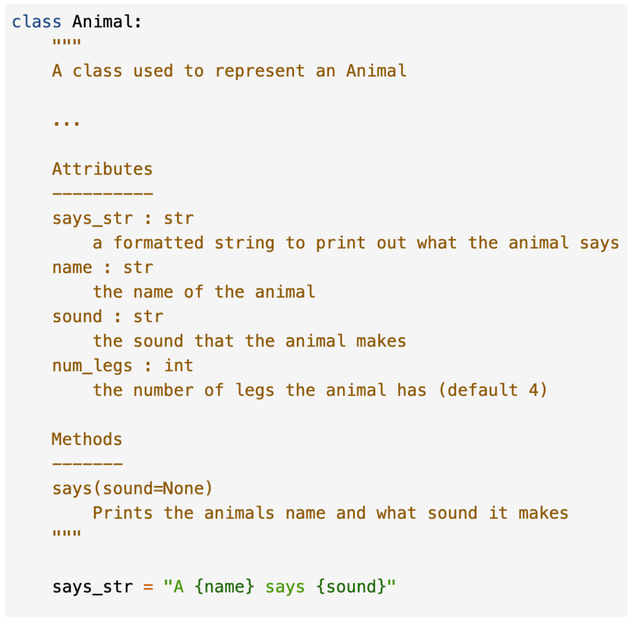
Public methods

```python
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    ----------
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -------
    says(sound=None)
        Prints the animals name and what sound it makes
    """

    says_str = "A {name} says {sound}"
```

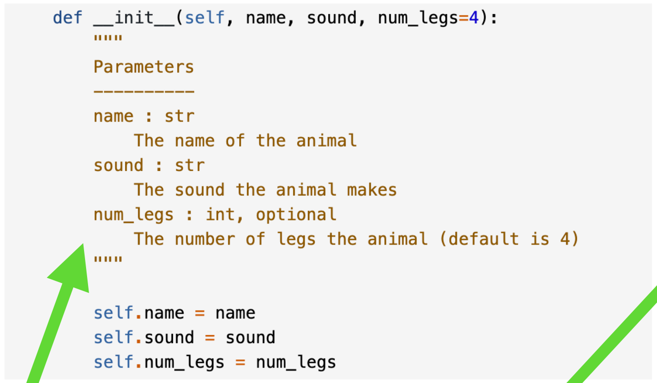
Potentially: Any info for subclasses

Docstrings

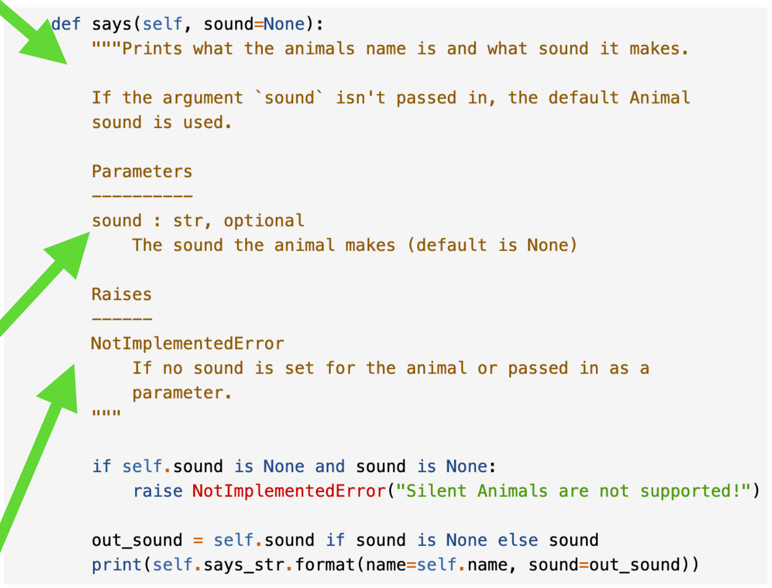
Brief description of purpose and functionality



```python
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    ----------
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -------
    says(sound=None)
        Prints the animals name and what sound it makes
    """

    says_str = "A {name} says {sound}"
```



```python
    def __init__(self, name, sound, num_legs=4):
        """
        Parameters
        ----------
        name : str
            The name of the animal
        sound : str
            The sound the animal makes
        num_legs : int, optional
            The number of legs the animal (default is 4)
        """

        self.name = name
        self.sound = sound
        self.num_legs = num_legs
```



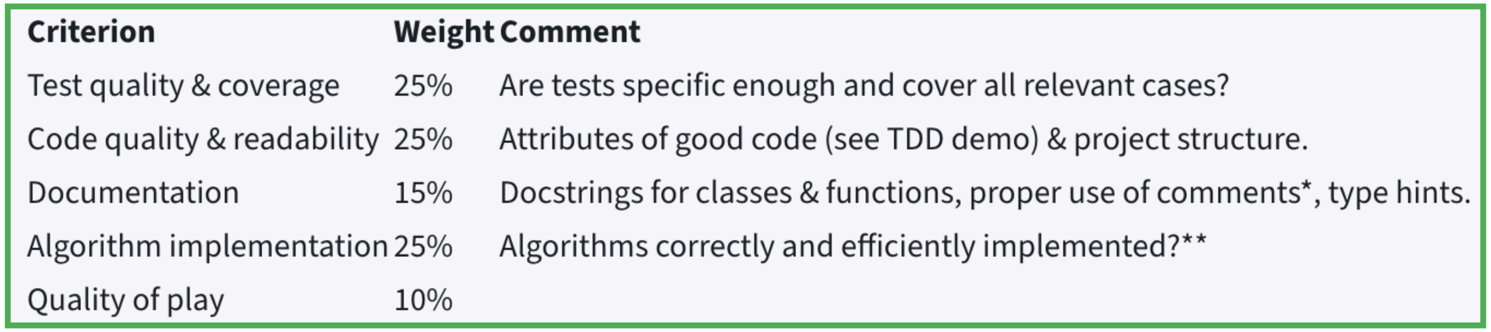
```python
def says(self, sound=None):
    """Prints what the animals name is and what sound it makes.

    If the argument `sound` isn't passed in, the default Animal
    sound is used.

    Parameters
    ----------
    sound : str, optional
        The sound the animal makes (default is None)

    Raises
    ------
    NotImplementedError
        If no sound is set for the animal or passed in as a
        parameter.
    """

    if self.sound is None and sound is None:
        raise NotImplementedError("Silent Animals are not supported!")

    out_sound = self.sound if sound is None else sound
    print(self.says_str.format(name=self.name, sound=out_sound))
```

Any arguments (label optional ones or defaults)

Exceptions raised

Potentially: Side effects & restrictions

Info for Submission



Criterion	Weight	Comment
Test quality & coverage	25%	Are tests specific enough and cover all relevant cases?
Code quality & readability	25%	Attributes of good code (see TDD demo) & project structure.
Documentation	15%	Docstrings for classes & functions, proper use of comments*, type hints.
Algorithm implementation	25%	Algorithms correctly and efficiently implemented?**
Quality of play	10%	

- Main goals:
  - Clean, readable, maintainable code
  - Tested code
  - Documented code
- Tests:
  - 'Coverage'
  - specific
  - one assertion per test function
- (**) 'Efficiently': No unnecessary steps, clean, reasonable heuristic.
  No strict performance criterion.
- 40/100 total points

- Quality of play is more a bonus, is heavily dependent on the other categories
- Please no copied minimax/negamax/MCTS pseudocode
- Keep attributes of good code and code smells in mind
- Documentation:
  - Docstrings for functions and classes, I recommend Google's or Numpy/SciPy's format
  - Type hints: signature required, docstring optional
  - (*) Comments: Make an effort to avoid them!