Don't repeat yourself

CREATING ROBUST WORKFLOWS IN PYTHON

Martin Skarzynski

Co-Chair, Foundation for Advanced Education in the Sciences (FAES)

What will you learn?

Learning Objectives:

- Follow best practices
- Use helpful technologies
- Write Python code that is easy to
  - read
  - use
  - maintain
  - share

- Develop your own personal workflow
  - Steps you take
  - Tools you use
  - Can evolve over time

The DRY principle

```
# Read in dataset info from text files
with open('diabetes.txt', 'r') as file:
    diabetes = file.read()

with open('boston.txt', 'r') as file:
    boston = file.read()

with open('iris.txt', 'r') as file:
    iris = file.read()
```

- DRY (Don't Repeat Yourself)
- WET (Waste Everyone's Time)

Functions

One of the repetitive code blocks:

```
# Read in diabetes.txt
with open('diabetes.txt', 'r') as file:
    diabetes = file.read()
```

- `filename` parameter
  - represents any possible filename
- `"diabetes.txt"` argument
  - a specific filename

A function definition:

```
# Define a function to read text files
def read(filename):
    with open(filename, 'r') as file:
        return file.read()
```

A function call:

```
# Use read() to read in diabetes.txt
diabetes = read("diabetes.txt")
```

Repetitive function calls

```
# Define a function to read text files
def read(filename):
    with open(filename, 'r') as file:
        return file.read()

# Use read() to read text files
diabetes = read("diabetes.txt")
boston = read("boston.txt")
iris = read("iris.txt")
```

- Define a function
- One `with` statements instead of three
- Three repetitive function calls

List comprehensions

```
# Create a list of filenames
filenames = ["diabetes.txt",
             "boston.txt",
             "iris.txt"]

# Read files with a list comprehension
file_list = [read(f)
             for f
             in filenames]
```

- Avoid writing out each function call
- Use a list comprehension
  - Similar to a `for` loop:

```
# View file contents
for f in filenames:
    read(f)
```

Multiple assignment

```
# Create a list of filenames
filenames = ["diabetes.txt",
             "boston.txt",
             "iris.txt"]

# Read files with a list comprehension
file_list = [read(f)
             for f
             in filenames]

diabetes, boston, iris = file_list
```

- Use multiple assignment
- Unpack the list
- Into multiple variables

Multiple assignment

```
# Create a list of filenames
filenames = ["diabetes.txt",
             "boston.txt",
             "iris.txt"]

# Read files with a list comprehension
diabetes, boston, iris = [read(f)
                          for f
                          in filenames]
```

- Use multiple assignment
- Unpack the list comprehension
- Into multiple variables
- DRY code!

Standard library

```
from pathlib import Path

# Create a list of filenames
filenames = ["diabetes.txt",
             "boston.txt",
             "iris.txt"]

# Use pathlib in a list comprehension
diabetes, boston, iris = [
    Path(f).read_text()
    for f in filenames
]
```

- `read_text()` method
- `Path` class
  - Opens and closes file automatically
  - No need for `with` statements
  - Not a built-in object
  - Must be imported before use
- `pathlib` module
  - Standard library
  - Included with Python

Generator expressions

```
from pathlib import Path

# Create a list of filenames
filenames = ["diabetes.txt",
             "boston.txt",
             "iris.txt"]

# Use pathlib in a generator expression
diabetes, boston, iris = (
    Path(f).read_text()
    for f in filenames
)
```

- To turn a list comprehension
- Into a generator expression
- Replace square brackets: `[]`
- With parentheses: `()`
- Generator expression produce generators
- Generators
  - Keep track of generated values
  - Can run out of values

Summary

- Don't Repeat Yourself (DRY)
- Tools in our DRY toolbox:
  - Functions (e.g. `read()`)
  - Methods (e.g. `read_text()`)
  - `for` loops
  - List comprehensions
  - Generator expressions
  - Python standard library, e.g. `pathlib`

Let's practice writing DRY code!

CREATING ROBUST WORKFLOWS IN PYTHON

Modularity

CREATING ROBUST WORKFLOWS IN PYTHON

Martin Skarzynski

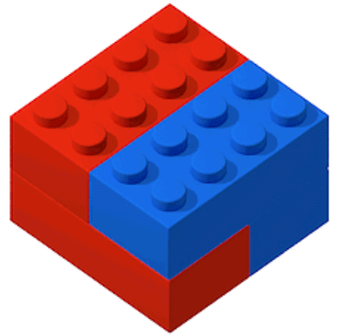
Co-Chair, Foundation for Advanced Education in the Sciences (FAES)

What is modularity?

- Independent, reusable objects
- Each object only has one job
- Separate code into modules and scripts

Modules and scripts

- Python code files
- `.py` extensions



Modules versus scripts

Modules

- Are imported
- Provide tools
- Define functions

The `say` module:

```
def hello():
    print("Hello World!")
```

Scripts

- Are run
- Perform actions
- Call functions

A script:

```
import say

say.hello()
```

Function definition and calls

```
def hello():
    print("Hello World!")

hello()
```

```
Hello World!
```

```
import say

say.hello()
```

```
Hello World!
Hello World!
```

Function definition and calls

```
def hello():
    print("Hello World!")

hello()
```

```
Hello World!
```

```
from say import hello

hello()
```

```
Hello World!
Hello World!
```

Module-script hybrid

```
def hello():
    print("Hello World!")

if __name__ == '__main__':
    hello()
```

```
Hello World!
```

```
from say import hello

hello()
```

```
Hello World!
```

The __name__ variable

```
def name():
    print(__name__)

if __name__ == '__main__':
    name()
```

```
__main__
```

When run as a script:

- `__name__` is `'__main__'`
- the `if` statement code block is run

```
import say

say.name()
```

```
say
```

When imported as a module:

- `__name__` is the module name
- the `if` statement code block is skipped

One function to rule them all

```
from pathlib import Path

def do_everything(filename, match):
    matches = (line for line in Path(filename).open() if match in line)
    flat = (string for sublist in matches for string in sublist)
    num_gen = (int(substring) for string in flat
               for substring in string.split() if substring.isdigit())
    return zip(num_gen, num_gen)
```

- Many responsibilities: obtain matches, extract numbers etc.

One job per function

```
def generate_matches(filename, match):
    return (line for line in Path(filename).open() if match in line)

def flatten(nested_list):
    return (string for sublist in nested_list for string in sublist)

def generate_numbers(string_source):
    return (int(substring) for string in string_source
            for substring in string.split() if substring.isdigit())

def pair(generator):
    return zip(generator, generator)
```

Iterators

```
def pair(items):
    iterator = iter(items)
    return zip(iterator, iterator)

pairs = list(pair([1, 2, 3, 4]))
pairs
```

```
[(1, 2), (3, 4)]
```

`iter()`

- turns its input (e.g. `list`)
- into an iterator (e.g. `list_iterator`)

```
type(iter([1, 2, 3, 4]))
```

```
list_iterator
```

Generators are iterators

```
def pair(items):
    iterator = iter(items)
    return zip(iterator, iterator)

pairs = list(pair([1, 2, 3, 4]))
pairs
```

```
[(1, 2), (3, 4)]
```

`iter()` has no effect on generators:

```
type(iter(x for x in [1, 2, 3, 4]))
```

```
generator
```

Adaptable functions

```
def pair(items):
    iterator = iter(items)
    return zip(iterator, iterator)

pairs = list(pair([1, 2, 3, 4]))
list(flatten(pairs))
```

```
[1, 2, 3, 4]
```

- Modular functions
  - Adaptable
  - Reusable
  - For example, `flatten()` can
    - Recreate the original list
    - From the `pairs` variable

Let's practice writing modular code!

CREATING ROBUST WORKFLOWS IN PYTHON

Abstraction

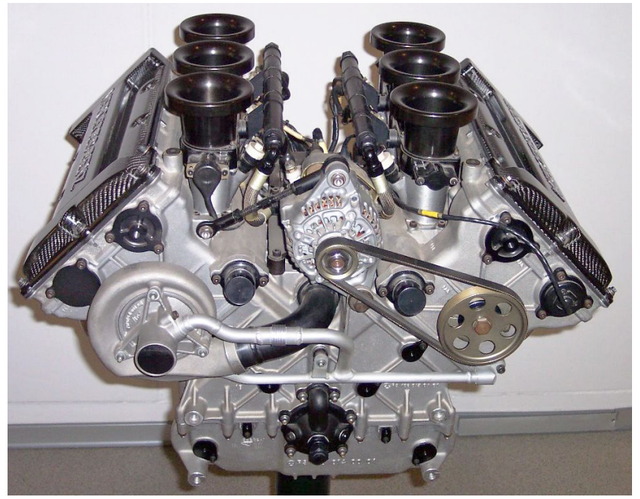
CREATING ROBUST WORKFLOWS IN PYTHON

Martin Skarzynski

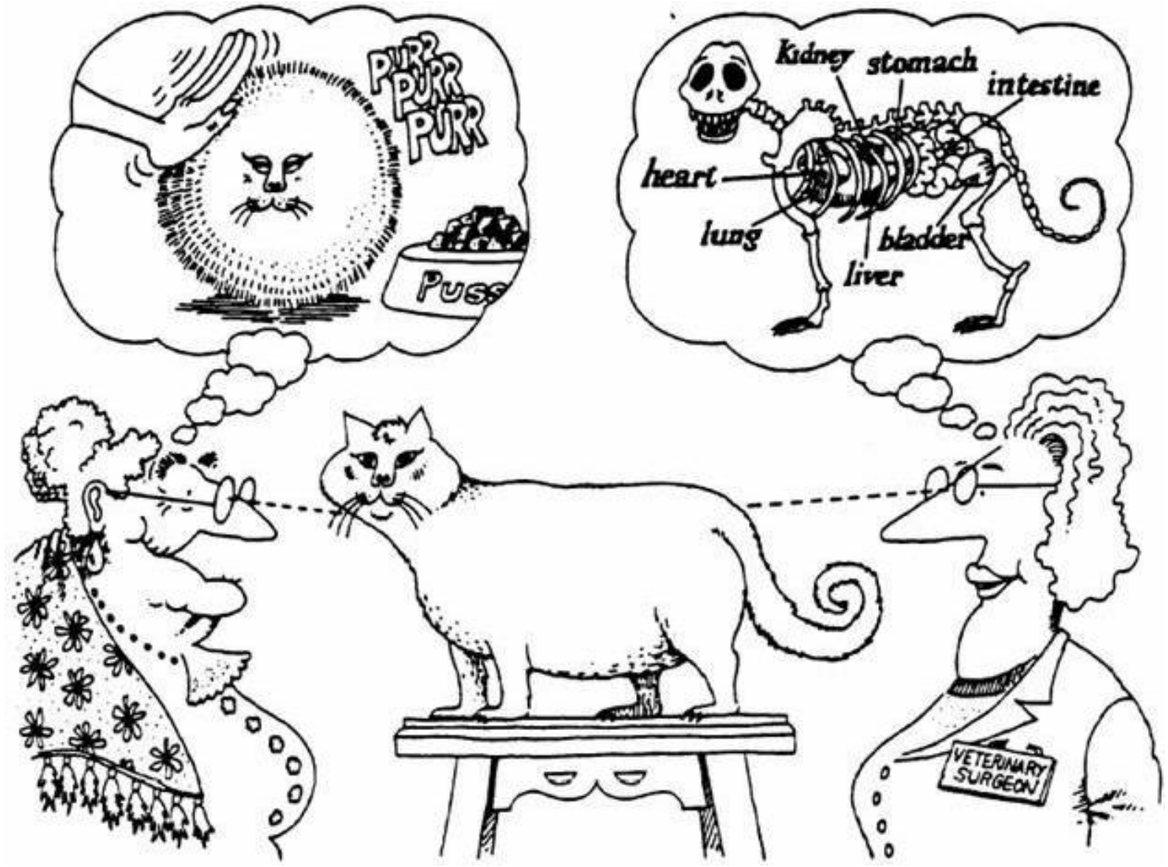
Co-Chair, Foundation for Advanced Education in the Sciences (FAES)

Abstraction

- Hide implementation details
- Design user interfaces
- Facilitate code use
- Car example:
  - Engine
    - Combustion
    - Electric



Classes



Booch, G. et al. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2007, p. 45.

- Templates for creating Python objects
- Represent real-life objects
- `Cat` class example:
  - User interface:
    - `feed()` and `rub()` methods
  - Implementation details:
    - Feline anatomy

Class definition

```
from pathlib import Path

class TextFile:
    def __init__(self, file):
        self.text = Path(file).read_text()
```

- The `TextFile` class
  - Represents any text file
  - Creates `TextFile` instances
    - Represent specific text files

Instantiation

```
diabetes = TextFile('diabetes.txt')

diabetes.text[:20]
```

```
'.. _diabetes_dataset'
```

- `TextFile` creates instances
  - By passing the `file` argument
  - To the `__init__()` method

```
def __init__(self, file):
    self.text = Path(file).read_text()
```

Instance attributes

```
from pathlib import Path

class TextFile:
    def __init__(self, filename):
        self.text = Path(filename).read_text()
        self.words = ''.join(c if c.isalpha() else ' ' for c in self.text).split()
```

Instance methods

```
from pathlib import Path

class TextFile:
    def __init__(self, filename):
        self.text = Path(filename).read_text()
        self.words = ''.join(c if c.isalpha() else ' ' for c in self.text).split()

    def len_dict(self):
        return {word: len(word) for word in self.words}
```

Method chaining

```
from pandas import DataFrame

(DataFrame(diabetes.len_dict().items())
 .sort_values(by=1, ascending=False)
 .head(n=4)
)
```

```
                 0   1
40  characteristics  15
17     measurements  12
31     quantitative  12
54      information  11
```

- `DataFrame` instance methods

```
def head(self, n=5):
  return self.iloc[:n]
```

  - Accept `DataFrame` instances
    - As their `self` argument
  - Return `DataFrame` instances
    - By returning `self`
  - Work well in method chains

Class attributes

```
class TextFile:

    instances = []

    def __init__(self, file):
        self.text = Path(file).read_text()
        self.__class__.instances.append(file)

TextFile.instances
```

```
[]
```

Class methods

```
class TextFile:

    instances = []

    def __init__(self, file):
        self.text = Path(file).read_text()
        self.__class__.instances.append(file)

    @classmethod
    def instantiate(cls, filenames):
        return (cls(filename) for filename in filenames)
```

Class methods

```
class TextFile:

    instances = []

    def __init__(self, file):
        self.text = Path(file).read_text()
        self.__class__.instances.append(file)

    @classmethod
    def instantiate(cls, filenames):
        return map(cls, filenames)
```

Instantiate

```
iris = TextFile('iris.txt')

boston, diabetes = TextFile.instantiate(['boston.txt', 'diabetes.txt'])

TextFile.instances
```

```
['iris.txt', 'boston.txt', 'diabetes.txt']
```

Let's practice defining classes and methods!

CREATING ROBUST WORKFLOWS IN PYTHON