PCL2-Tutorial 08

Lucas Suomela, Rong Li, Tosca Peruzzi-Vieli, Isabelle Cretton

Welcome!

Mid-term Review: Duck Typing

An application of the duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck"

A concept used in dynamic languages such as Python: if an object behaves like a certain type (e.g., it has the required methods and attributes), it is considered to be of that type.

In this example, we have a **Bird** class and an **Airplane** class, both of which have a **fly** method. The function **in_the_sky** doesn't care about the type of the object passed in; it only cares whether the object has a **fly** method. This is duck typing in action!

```python
class Bird:
    def fly(self):
        return 'Flap Flap!'

class Airplane:
    def fly(self):
        return 'Zoom Zoom!'

def in_the_sky(flier):
    print(flier.fly())

pigeon = Bird()
boeing = Airplane()
in_the_sky(pigeon)
in_the_sky(boeing)

# Output:
# 'Flap Flap!'
# 'Zoom Zoom!'
```

Mid-term Review: Variable Names

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive
- A variable name cannot be any of the Python keywords.

Examples of Valid Names:

Username
_user1
user_name2
User3
Return

Examples of Invalid Names:

2user
user-name
v3.9
class
for
return

Mid-term Review: Semantic Versioning

Semantic Versioning uses a three-part version number format:
MAJOR.MINOR.PATCH

- **MAJOR version** when you make incompatible API changes,
- **MINOR version** when you add functionality in a backwards-compatible manner,
- **PATCH version** when you make backwards-compatible bug fixes.

Example 1: A software library introduces a new feature that adds new APIs without affecting the existing ones.
Before: 3.2.5
After: 3.3.0 (Minor version incremented)

Example 2: A major release that changes the library's architecture, potentially breaking existing integrations.
Before: 1.0.0
After: 2.0.0 (Major version incremented)

Example 3: Shortly after a major release, a critical security vulnerability is discovered and promptly fixed.
Before: 5.0.0
After: 5.0.1 (Patch version incremented)

Mid-term Review: CI/CD Pipeline

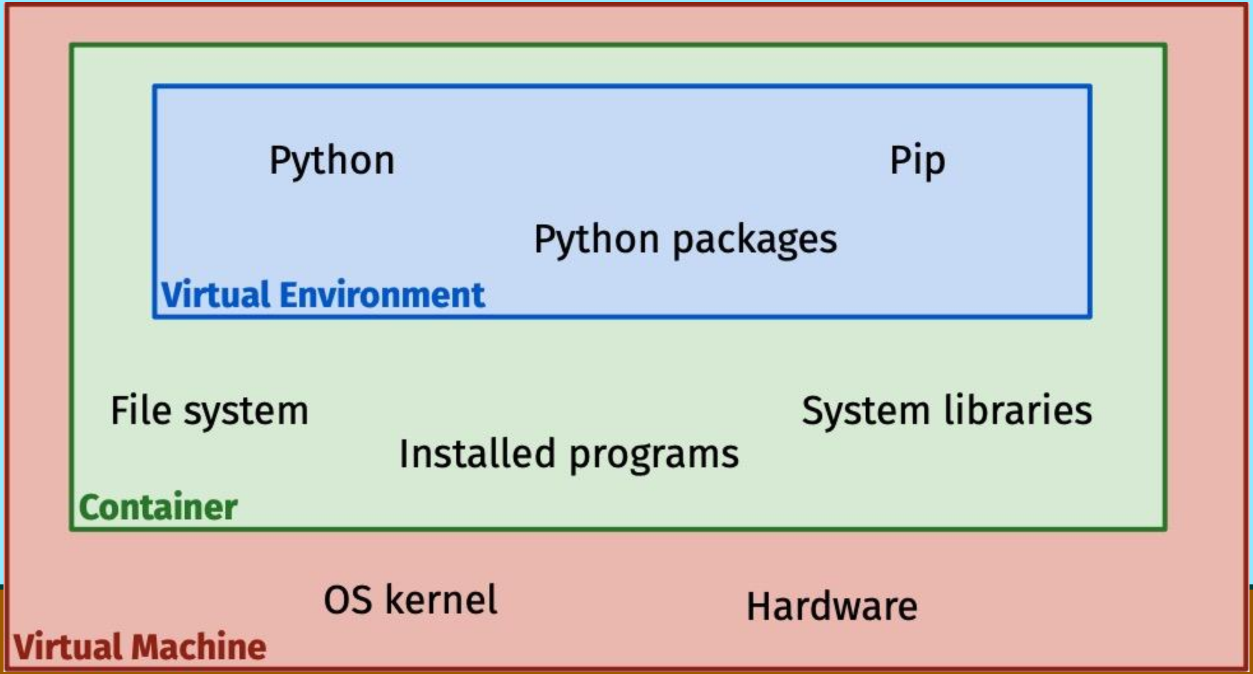
Continuous Integration (CI):
Continuously and frequently **merge small changes** into the main branch.
→ Developers can work independently and without complex merge conflicts.

Continuous delivery (CD):
Continuously and frequently **release new versions**.
→ More immediate testing and delivery of updates, less risk in releasing.

How it can be used to improve code quality?

- run unit tests and report coverage after each commit
- Linter → a program that checks syntax errors, undefined variables, formatting errors in your code
- Code formatter → a program to standardize code formatting
- AI tools → check your code for potential security vulnerabilities or bad programming patterns.

Mid-term Review: Container



Containerization is a lightweight form of virtualization that involves encapsulating an application and its dependencies into a container that can be executed consistently across various computing environments.

Containers are isolated environments including programs, libraries, and the file system. However, they use the host's OS kernel, making them lighter and more resource-efficient than **VMs**.

Mid-term Review

And some pages from our previous tutorials...

Mid-term Review: References

Think back to immutable vs. mutable data

- A variable name references to where the data is located in memory
- If the underlying object is mutable, then any modifications done will persist
- If the underlying object is immutable, modifications will not persist.

How References Work

- Assignment of objects to variables creates a reference, not a copy.
- Multiple variables can reference the same object.

```
# Assigning a list to a variable
original_list = [1, 2, 3]
# Creating a reference to the same list
reference_list = original_list

# Modifying the original list
original_list.append(4) # this will also change the reference
```

```
print(original_list)
# output: [1, 2, 3, 4]
```

```
print(reference_list)
# output: [1, 2, 3, 4]
```

Mid-term Review: Namespaces

- **A namespace is a container (dictionary) where names are mapped to objects, such as variables and functions**
- **Enables the Python interpreter to distinguish between identifiers with the same name but in different namespaces**
- **Scope Resolution: The Process of accessing variables across different namespaces**
- **Follows LEGB rule: Local -> Enclosed -> Global -> Built-in**

- Variables in Python are names and belong to exactly one namespace
- Modify namespaces dynamically by adding, changing or removing names

Modules, classes, functions and methods have their own local namespace.

Mid-term Review: Static vs. Class Methods

A **static method** does not receive an implicit first argument (self for instance or cls for class). A static method is also a method that **is bound to the class** and not the instance of the class. They are used when some functionality is related to the class but does not need to **access or modify** the class's state.

```
class Geometry:
    @staticmethod
    def area_of_triangle(base, height):
        return 0.5 * base * height

# Use case
print(Geometry.area_of_triangle(10, 20))
```

A **class method** is bound to the class and not the instance of the class. They can modify the class state that applies across all instances of the class, via the cls argument, which is a reference to the class itself.

```
class Counter:
    count = 0

    @classmethod
    def increment(cls):
        cls.count += 1
        return cls.count

# Use case
Counter.increment()  # count is now 1
Counter.increment()  # count is now 2
```

Class method vs Static Method

- A class method takes cls as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- We use @classmethod decorator to create a class method and we use @staticmethod decorator to create a static method.

Mid-term Review: Iterable vs. Iterator

An **iterable** is an object which can be looped over or iterated over in a loop. Examples of iterables include **lists, sets, tuples, dictionaries, strings**, etc. Any object that has an __iter__() method which returns an **iterator**, or a __getitem__() method that can take sequential indexes, is an iterable.

An **iterator** is an object that allows you to iterate over collections of data, consisting of the methods __iter__() and __next__(). The __iter__() returns the iterator object itself and is used once; the __next__() method returns the next value from the iterator. When there are no more items, __next__() raises a **StopIteration** exception.

```
my_list = [1, 2, 3, 4] # a list is an iterable
my_iterator = iter(my_list)  # Or my_list.__iter__()

while True:
    try:
        item = next(my_iterator)  # Or my_iterator.__next__()
        print(item)
    except StopIteration:
        break
```

Mid-term Review: Generator

Generators are a type of **iterator**, but they are written as regular functions and use the yield statement to produce a series of values lazily, meaning they generate values on the fly and only when needed. This makes them memory-efficient, especially useful for large datasets.

Generator expressions provide a concise way to create generators. They create a generator that yields items on-the-fly, which makes them highly memory-efficient, especially useful for large datasets.

Syntax: similar to list comprehension except it uses parentheses ():
(expression **for** item **in** iterable **if** condition)

Let's see an example of this generator function:

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
for num in counter:
    print(num)
```

Another example about generator expressions:

```
infile = open("very_large_file.txt")
 generator = (line.strip().upper() for line in infile if line != "\n")
next(generator)
```

Mid-term Review: Decorator

Decorators are **higher-order functions** used to modify the behavior of a function. It allows you to add new functionality to an existing object without modifying its structure. Decorators are usually called **before** the definition of a function you want to decorate.

A **higher order function** is a function that takes a function as an argument or that returns a function.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

my_decorator is a function that takes another function (say_hello) and extends its behavior without explicitly modifying say_hello's definition.

Mid-term Review: Pure Functions

A **pure function** is a function that satisfies two main conditions:

- **Deterministic Output:** For the same input values, always returns the same result.
- **No Side Effects:** Does not alter any external state or data outside of its scope. Does not print anything.

Why do we need pure functions?

- Since they always return the same result for the same inputs, they are straightforward to test.
- Their isolated nature makes them easily reusable and predictable.

The increment() method is impure because:

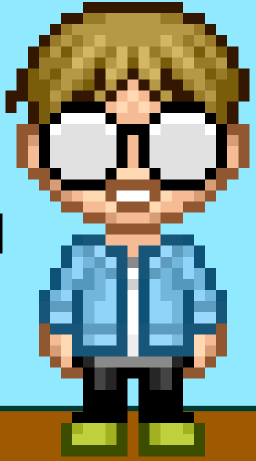
- It modifies a global variable, affecting external state.
- Repeated calls with the same external inputs (no inputs in this case) produce different results.

The add() method is pure because:

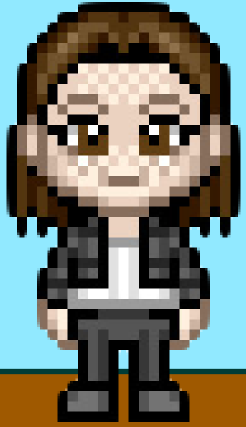
- It always returns the same result for the same inputs.
- It does not modify any internal or external state of the class.

```
number = 0

class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        global number
        number += 1
        return number

    def add(self, x, y):
        return x + y

# Use case
counter = Counter()
print(counter.increment())  # Output: 1
print(counter.increment())  # Output: 2

print(counter.add(5, 3))  # Output: 8
print(counter.add(5, 3))  # Output: 8
```

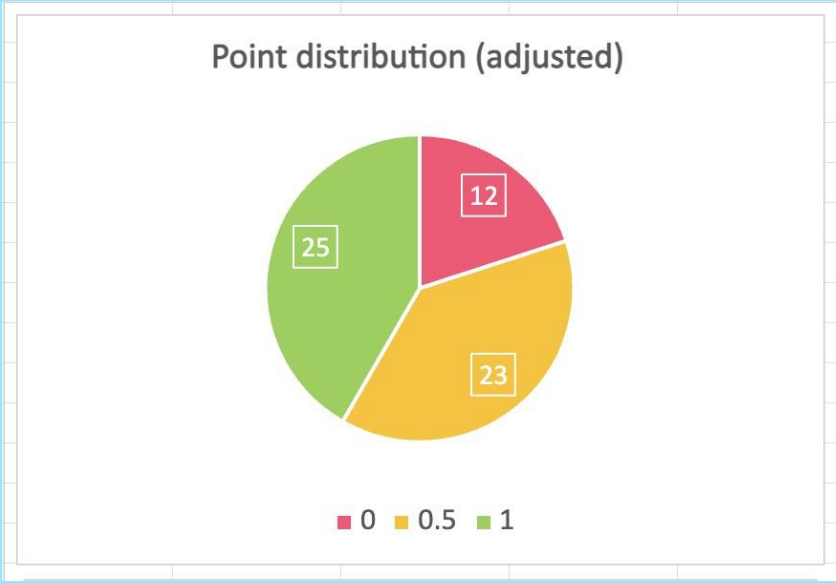
Mid-term Review

Q&A TIME

Exercise 6: feedback



- We grade in half points, 0 - 0.5 - 1.
- Important to **run code** before submission, too many “simple” errors that would have been easy to spot when running scripts.
- If you don’t know how to run your scripts with arguments from the command line, we have a problem.
- Tests script are non-exhaustive, this has been mentioned in almost all previous exercises. Passing the provided test does not mean that the code meets the requirements set in the exercise description.

Some comments:

- 22 people did not release the assignment to us!
- Average points per person: 0.61.

Data Scaling: Sequential Text Processing

We must take data scaling into consideration when the amount of data is not compatible with traditional data processing methods.

- Use **generators** to read and write to files
- Use **fixed-size chunking** to arbitrary portion large text files while parsing them

You don't have to add a flag when using chunking!

```
with open('large_file.txt') as stream:
    # Call a generator function to parse the chunks
    pass
```

Using sequential text processing is excellent when you want to use `nltk` or `spaCy` with large text files

Refer to the excellent jupyter notebook from lecture 10!

Data Scaling: `lxml`

```python
def clean_and_yield_titles(xml):
    root = etree.fromstring(xml)
    # Create a list of books to avoid modification of the tree while iterating
    books = root.findall('book')
    for book in books:
        year = int(book.find('year').text)
        if year < 2000:
            # Clear the contents of the book element and then remove it from the root
            book.clear()
            # Remove the cleared element from the tree
            root.remove(book)
        else:
            # Yield the title of the book if the year is 2000 or later
            yield book.find('title').text
        # (Optional) delete the book
        del book
```

What's the mistake here? Why is this not efficient?

Data Scaling: `ijson`

```python
def extract(json_file):
    '''Get authors from a JSON file Containing books'''
    for x in ijson.items(open(json_file), 'item.author'):
        yield x
```

See the documentation here

Exercise 08

- Setup git-lfs
- Implement a JSON to XML converter
- Take memory- and time-efficiency into consideration
- Make the whole converter program into a Command-Line interface

Now It's Your Turn…

To-Do

- ❑ Work on exercise 8
- ❑ Ask us questions if you are confused!
- ❑ Buy an absurd amount of ram so you don't have to care about data scaling