Design Patterns Singleton

Note on Python: Class vs Instance Variables

- Class Variables — Declared inside the class definition (but outside any of the instance methods).
  - They are not tied to any particular object of the class.
  - Shared across all the objects of the class.
  - Modifying a class variable affects all objects instance at the same time.
- Instance Variable — Declared inside the constructor method of class (the __init__ method).
  - They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.
- Code Example

Advanced Python: Class vs Instance vs Static methods

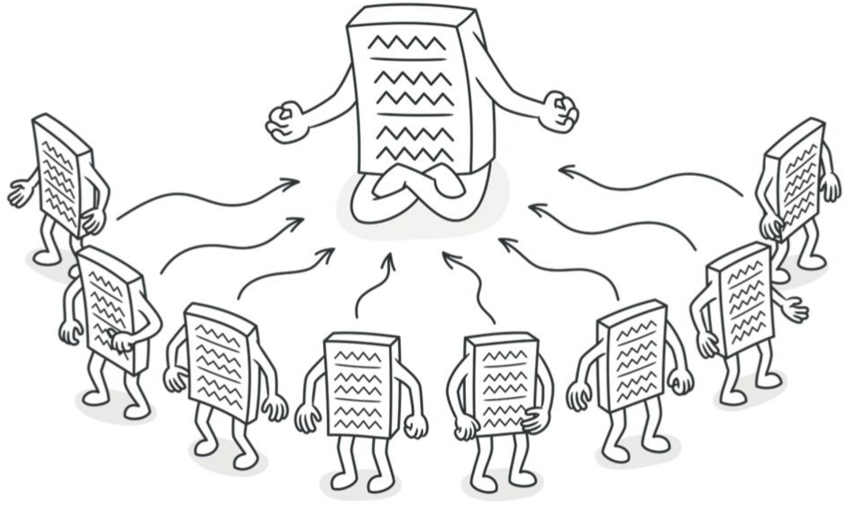
- Class methods don't need a class instance.
    - They can't access the instance (**self**) but they have access to the class itself via **cls**.
- Static methods don't have access to cls or self .
    - They work like regular functions but belong to the class's namespace.
- Code example

Expert Python: Metaclasses

- In the same way that a class functions as a template for the creation of objects, a metaclass functions as a template for the creation of classes.
- Metaclasses are sometimes referred to as class factories.
- A metaclass in Python is a class of a class that defines how a class behaves.
- A class is itself an instance of a metaclass

Singleton Design Pattern

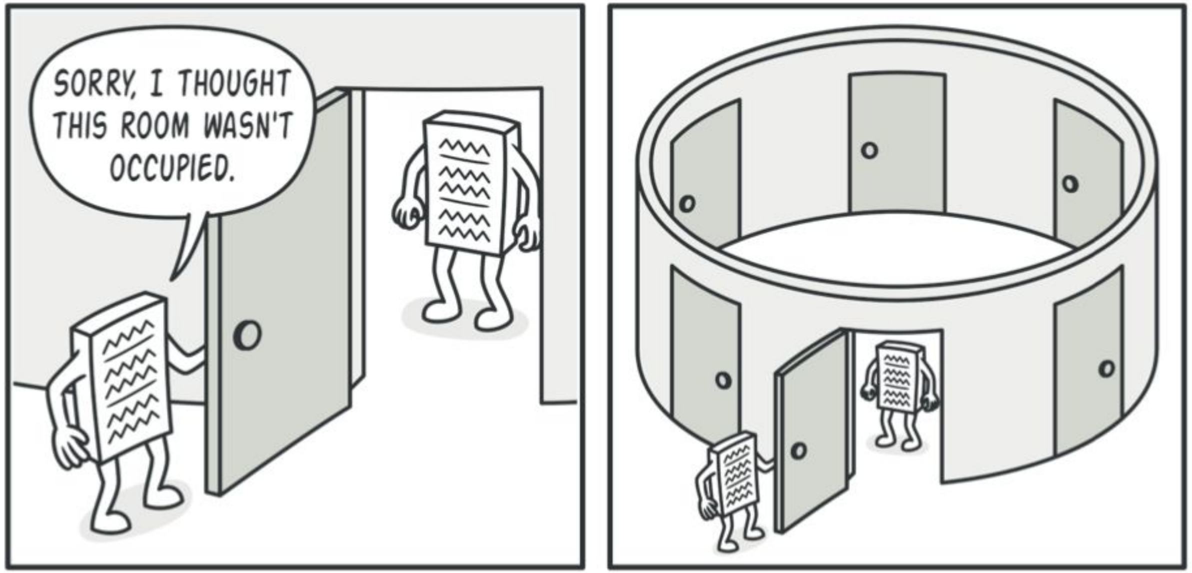
- Singleton is a creational design pattern
- Lets you ensure that a class has only one instance, while providing a global access point to this instance.



Singleton

- The Singleton pattern ensure that a class has **just a single instance.**
- The most common reason for this is to control access to some shared resource—for example, a database or a file.
- Here’s how it works:
    - imagine that you created an object, but after a while decided to create a new one.
    - Instead of receiving a fresh object, you’ll get the one you already created.
- Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

Singleton



Clients may not even realize that they're working with the same object all the time.

Singleton

- Provides a global access point to that instance.
- Remember those global variables that you (all right, me) used to store some essential objects?
    - While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.
- Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program.
- However, it also protects that instance from being overwritten by other code.

Singleton

The singleton design pattern is useful when:

- you need to create only one, heavyweight, object (ML model, etc)
- you need some sort of object capable of maintaining a global state for your program.

Other possible use cases are:

- Controlling concurrent access to a shared resource. For example, the class managing the connection to a database.
- A service or resource that is transversal in the sense that it can be accessed from different parts of the application or by different users and do its work. For example, the class at the core of the logging system or utility.

Singleton: Real-World Analogy

- The government is an excellent example of the Singleton pattern.
- A country can have only one official government.
- Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge.

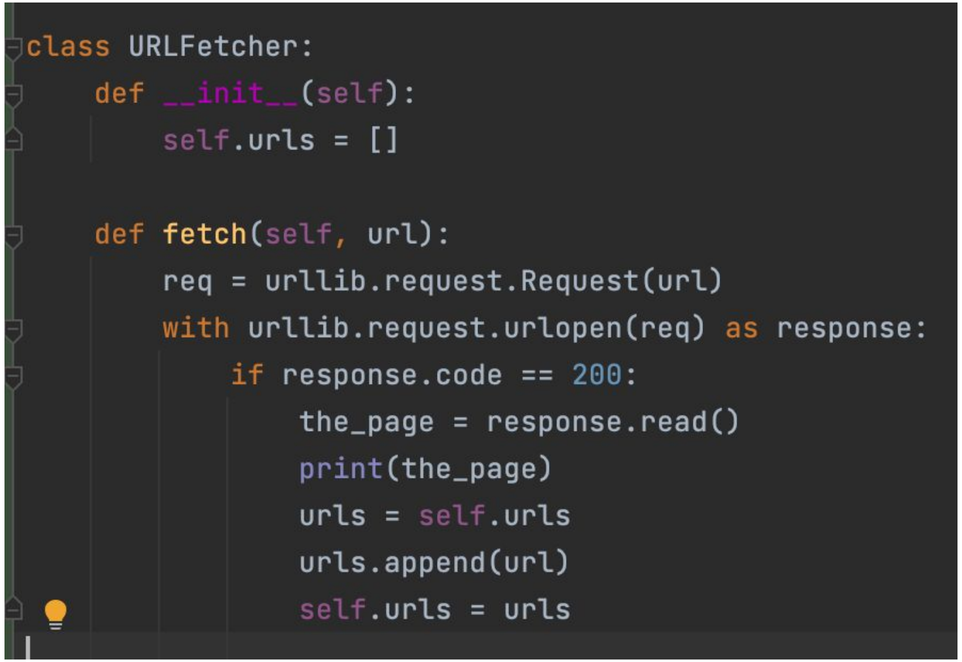
Singleton: Example

- Let's implement a program to fetch content from web pages.

- We will use the urllib module to connect to web pages using their URLs
    - the core of the program would be the URLFetcher class that takes care of doing the work via a fetch() method.
- We want to be able to track the list of web pages that were tracked, hence the use of the singleton pattern: we need a single object to maintain that global state.

Singleton: Example

First, our naive version, to help us track the list of URLs that were fetched, would be:

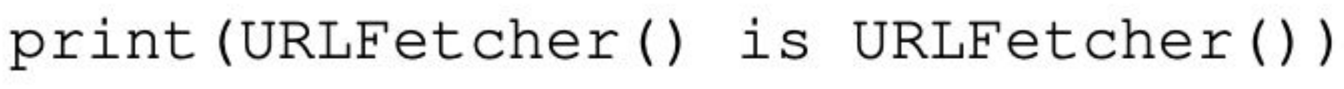


```python
class URLFetcher:
    def __init__(self):
        self.urls = []

    def fetch(self, url):
        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                the_page = response.read()
                print(the_page)
                urls = self.urls
                urls.append(url)
                self.urls = urls
```

Does that example create a singleton?

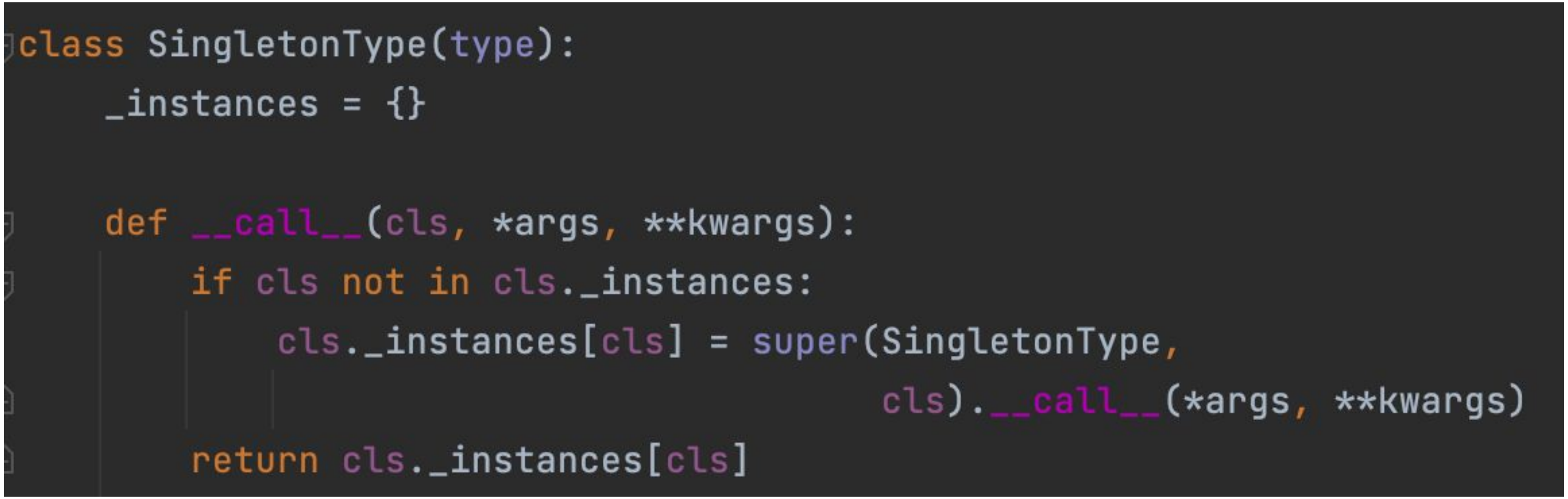
What will happen if we run



```
print(URLFetcher() is URLFetcher())
```

Singleton in Python3

We first implement a metaclass for the singleton, meaning the class (or type) of the classes that implement the singleton pattern, as follows:



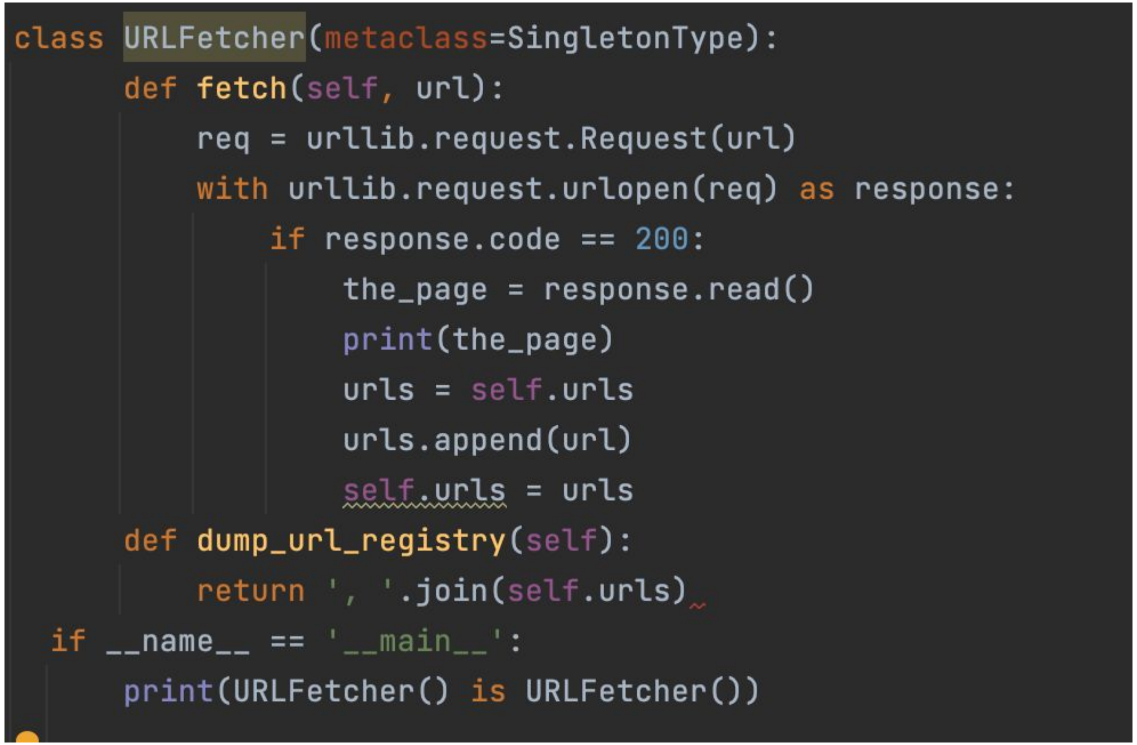
```python
class SingletonType(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(SingletonType,
                                        cls).__call__(*args, **kwargs)
        return cls._instances[cls]
```

Singleton in Python3

Now, we will rewrite our URLFetcher class to use that metaclass.

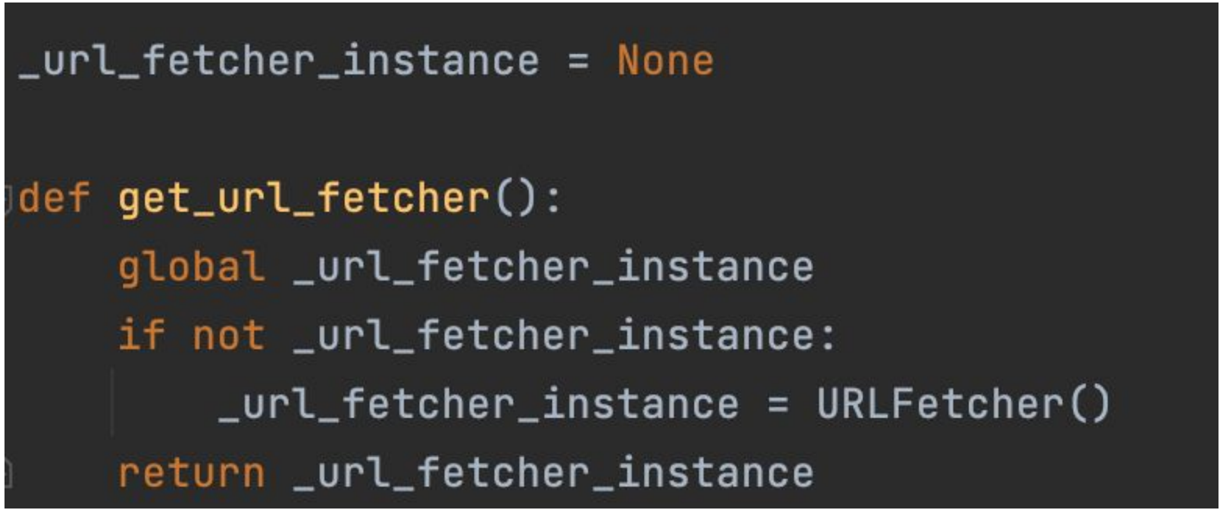
We also add a dump_url_registry() method, which is useful to get the current list of URLs tracked



```
class URLFetcher(metaclass=SingletonType):
      def fetch(self, url):
          req = urllib.request.Request(url)
          with urllib.request.urlopen(req) as response:
              if response.code == 200:
                  the_page = response.read()
                  print(the_page)
                  urls = self.urls
                  urls.append(url)
                  self.urls = urls
      def dump_url_registry(self):
          return ', '.join(self.urls)
  if __name__ == '__main__':
      print(URLFetcher() is URLFetcher())
```

Singleton in Python2 or without control of source

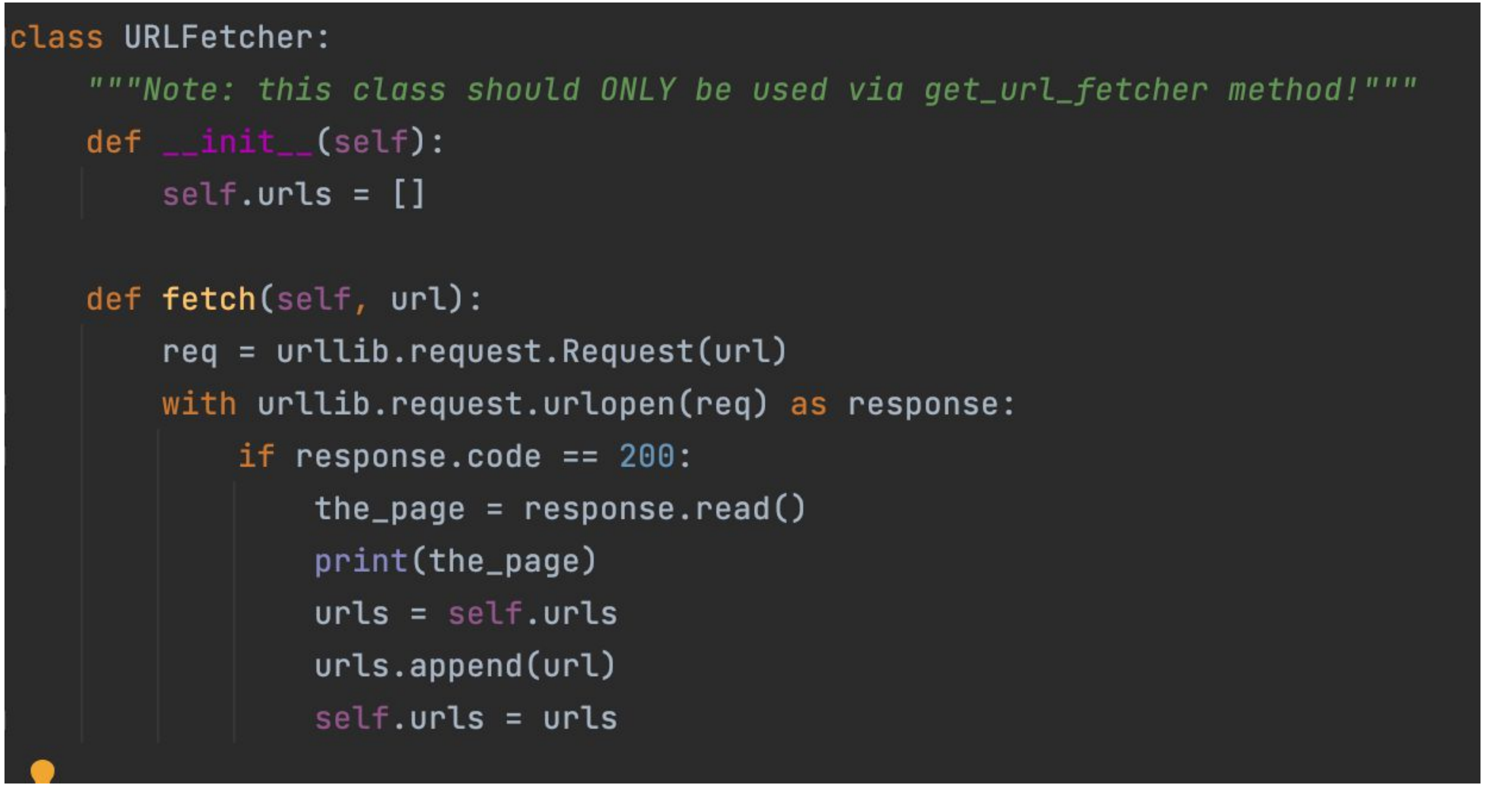
In Python2, the metaclass doesn’t exist yet. Instead, we add a global var to hold the singleton and refer to it via a method to get the singleton.



```
_url_fetcher_instance = None

def get_url_fetcher():
    global _url_fetcher_instance
    if not _url_fetcher_instance:
        _url_fetcher_instance = URLFetcher()
    return _url_fetcher_instance
```

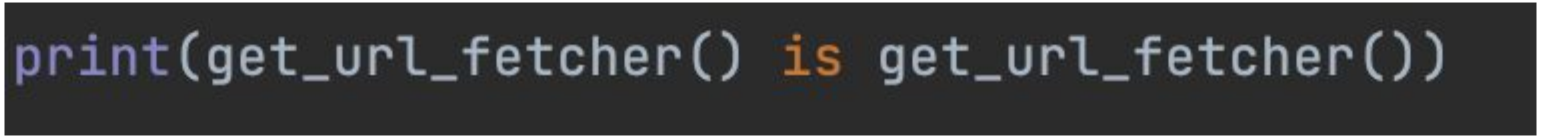
Singleton in Python2



```python
class URLFetcher:
    """Note: this class should ONLY be used via get_url_fetcher method!"""
    def __init__(self):
        self.urls = []

    def fetch(self, url):
        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                the_page = response.read()
                print(the_page)
                urls = self.urls
                urls.append(url)
                self.urls = urls
```

Does this work?



```
print(get_url_fetcher() is get_url_fetcher())
```

Singleton: Pros

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it’s requested for the first time.

Singleton: Cons

- If the object has some state that is SETTABLE, it’s a warning sign that Singleton isn’t the best option
- If some other totally unrelated process can change the object I’m using and affect my runtime behavior...just make sure that’s desired

Prototype Design Pattern

- A prototype design pattern is used for creating **exact copies of objects**
- This used to be hard/manual in Java and an entire pattern was set for it!
- Python does it automatically with the `copy` library

Prototype and python object copying

- The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):
  - A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
  - A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Here are the docs!