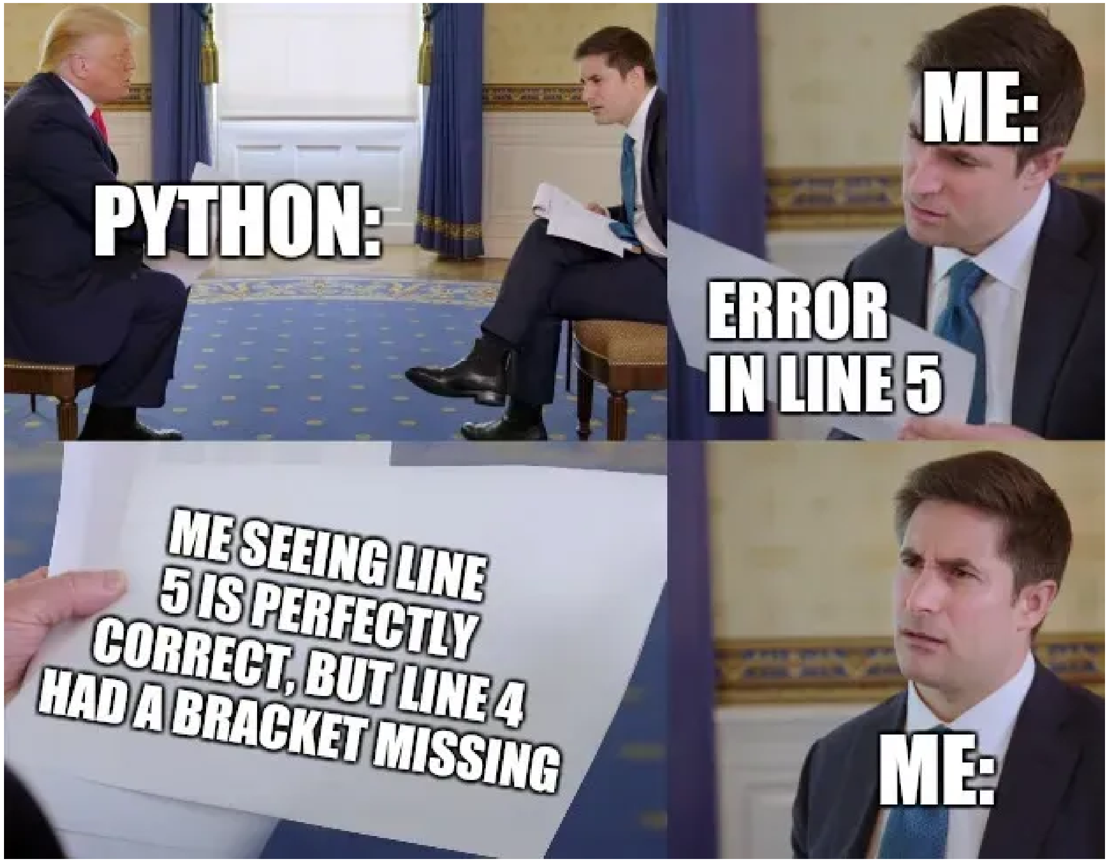
Discussion 8

DSC 20, Winter 2024

Meme of the Week


PYTHON:
ME:
ERROR IN LINE 5
ME SEEING LINE 5 IS PERFECTLY CORRECT, BUT LINE 4 HAD A BRACKET MISSING
ME:

Agenda

- Classes
- instance methods
- class methods
- static methods
- Inheritance
- super

Classes

Classes aggregate code together into a functional body. A lot of effective python code is written as classes (every time you install a new package, it's basically written as a lot of different classes). For example, recall the pandas dataframe. If you look at the source code for pandas, you'll see that it's a very complicated class definition.

Classes all generally have (at the minimum) a constructor (__init__) function and other object related methods that are used.

In [7]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    pass
```

Constructors

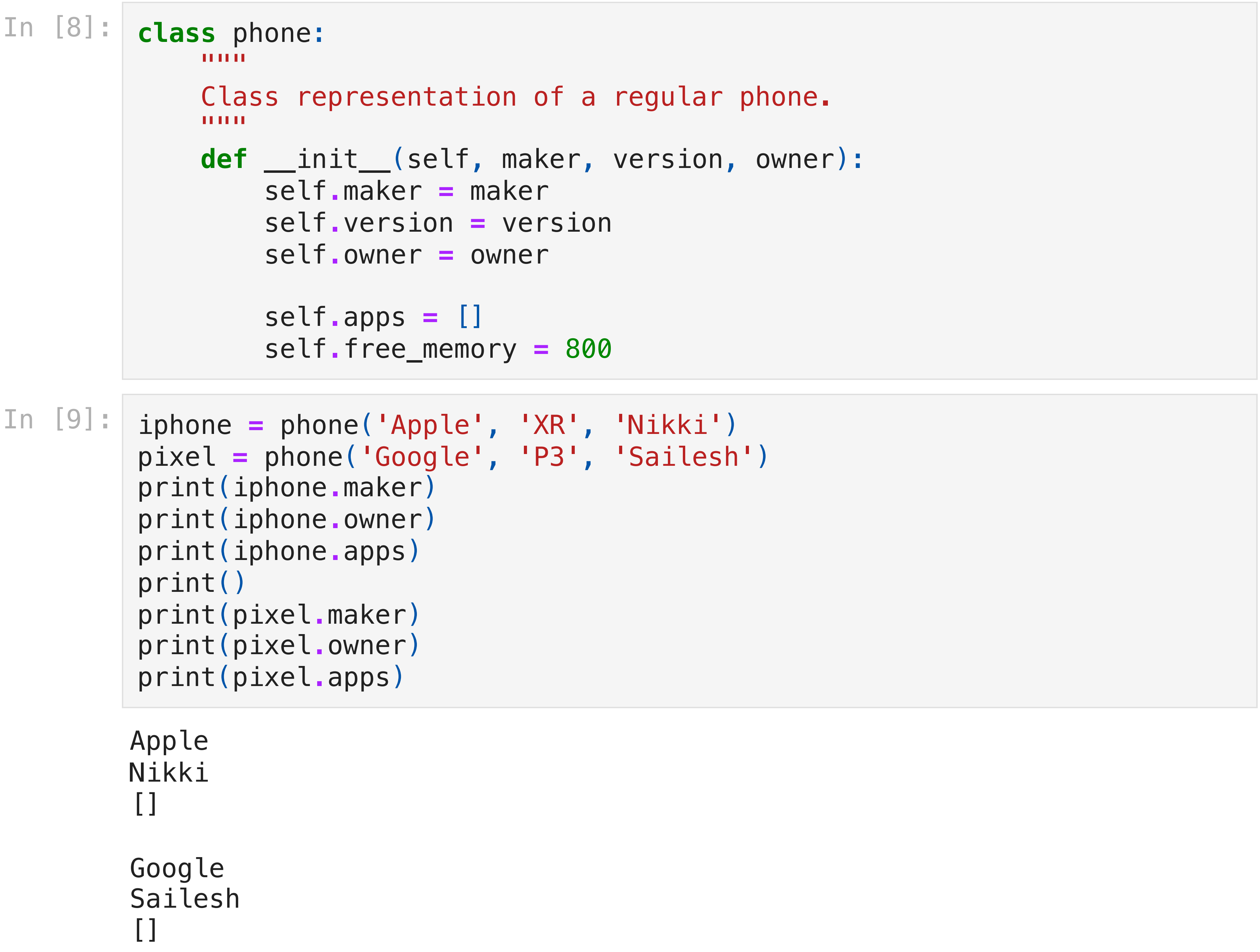
- denoted by __init__ (special function)
- should ingest self as the first parameter (ALWAYS)
- populates instance with attributes

Instances

When an object is created from a class, each individual unit is referred to as an instance. Think of it as "one of x" (for example, an instance of the iPhone class is "1/my iPhone")

Working with self

- Think of classes as a blueprint and everytime you create an instance of one, you've produced a "physical" manifestation.
- We use the self keyword to interact with THIS specific instance of the class.
- Try to think of 'self' as referencing a **specific, singular** instance of the class. Every method that works with a specific instance's values must be passed in self as an argument.



In [8]:

```python
class phone:
    """
    Class representation of a regular phone.
    """
    def __init__(self, maker, version, owner):
        self.maker = maker
        self.version = version
        self.owner = owner

        self.apps = []
        self.free_memory = 800
```

In [9]:

```python
iphone = phone('Apple', 'XR', 'Nikki')
pixel = phone('Google', 'P3', 'Sailesh')
print(iphone.maker)
print(iphone.owner)
print(iphone.apps)
print()
print(pixel.maker)
print(pixel.owner)
print(pixel.apps)
```

```
Apple
Nikki
[]

Google
Sailesh
[]
```

Instance vs Class variables

Instance variables are attached to instances of a class by keyword self (usually done in the constructor). Class variables are variables attached to the class itself.

Instance Methods

Instance methods operate on an instance of the class (an object), allowing the method to read or modify the state of the instance. Each instance method automatically takes `self` as its first argument, which is a reference to the instance on which the method was called.

- **Access to Instance and Class Data:** Instance methods can modify the object's state by accessing or updating instance attributes.
- **Usage:** Primarily used to define the behaviors of an object. Most operations involving objects of a class will use instance methods because they can access and modify the data within those objects.

In [12]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    id_num = 0
    def __init__(self, maker, version, owner):
        self.maker = maker
        self.version = version
        self.owner = owner
        self.id = phone.id_num
        phone.id_num+=1

        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        if self.free_memory - memory >= 0:
            self.apps.append(app)
            self.free_memory -= memory
            return True
        return False
```

In [13]:

```
iphone = phone('Apple', 'XR', 'Nikki')
print(iphone.install_app('duolingo', 600))
print(iphone.install_app('genshin impact', 1000))
print(iphone.apps)
```

```
True
False
['duolingo']
```

Checkpoint

Given the following code, what is the result of the final expression?

In [3]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    id_num = 0
    def __init__(self, owner):
        self.id = phone.id_num
        phone.id_num+=1
        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        if self.free_memory - memory >= 0:
            self.apps.append(app)
            self.free_memory -= memory
            return True
        return False
```

In [4]:

```
phone1 = phone('Anish')
phone2 = phone('TQ')
```

In [ ]:

```
phone.id_num
```

A. 0

B. 1

C. 2

D. 3

Checkpoint Solution

```
In [5]: phone.id_num
```

```
Out[5]: 2
```

Checkpoint

Given the following code, what is the result of the final expression?

In [ ]:

```python
class phone:
    """
    Class representation of a regular phone.
    """
    id_num = 0
    def __init__(self, owner):
        self.id = phone.id_num
        phone.id_num+=1
        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        if self.free_memory - memory >= 0:
            self.apps.append(app)
            self.free_memory -= memory
            return True
        return False
```

In [7]:

```python
phone1 = phone('Anish')
phone2 = phone('TQ')
phone1.install_app('tiktok', 1000);
```

In [ ]:

```python
phone1.free_memory == phone2.free_memory
```

A. True

B. False

Checkpoint Solution

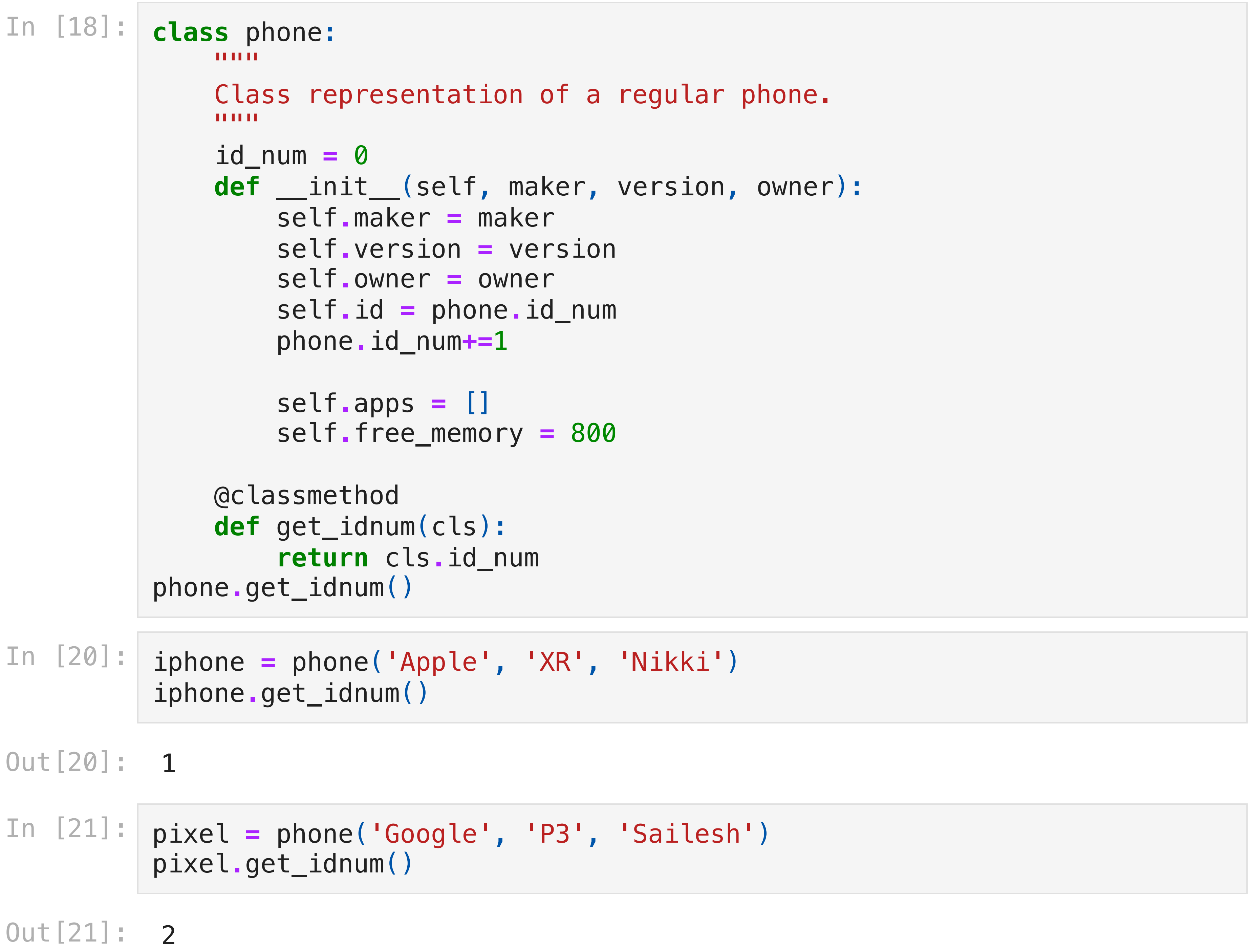
```
In [9]:  phone1.free_memory == phone2.free_memory

Out[9]:  True
```

@classmethod

The `@classmethod` decorator modifies a method to become a class method. Class methods affect the class itself and are called on the class rather than its instances. This means that they receive the class as their first argument (`cls`) instead of the instance (`self`).

- **Class-Level Operation:** `@classmethod` allows a method to modify class state that applies across all instances or to call other class methods.

- **Access to Instance Variables:** While class methods can access class attributes, they do not have direct access to instance variables unless explicitly passed an instance. This is because they are not bound to a specific instance but to the class itself.



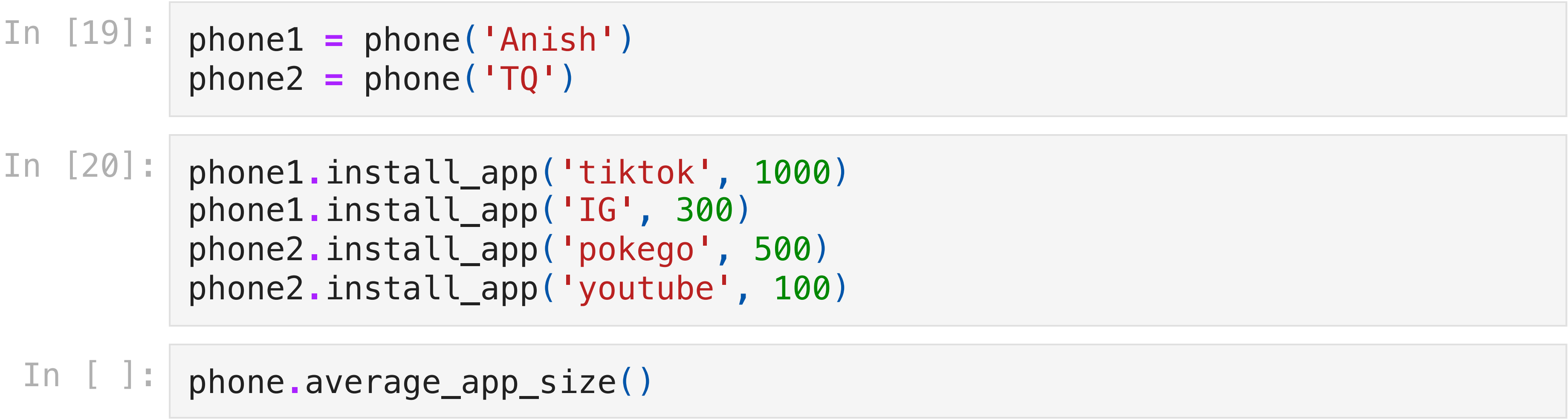
```
In [18]: class phone:
             """
             Class representation of a regular phone.
             """
             id_num = 0
             def __init__(self, maker, version, owner):
                 self.maker = maker
                 self.version = version
                 self.owner = owner
                 self.id = phone.id_num
                 phone.id_num+=1

                 self.apps = []
                 self.free_memory = 800

             @classmethod
             def get_idnum(cls):
                 return cls.id_num
         phone.get_idnum()

In [20]: iphone = phone('Apple', 'XR', 'Nikki')
         iphone.get_idnum()

Out[20]: 1

In [21]: pixel = phone('Google', 'P3', 'Sailesh')
         pixel.get_idnum()

Out[21]: 2
```

Checkpoint

Given the following code, what is the result of the final expression?

In [32]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    total_memory_used = 0
    total_apps_installed = 0

    def __init__(self, owner):
        self.owner = owner
        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        if self.free_memory >= memory:
            self.apps.append(app)
            self.free_memory -= memory
            phone.total_memory_used += memory
            phone.total_apps_installed += 1

    @classmethod
    def average_app_size(cls):
        if cls.total_apps_installed == 0:
            return 0
        return cls.total_memory_used / cls.total_apps_installed
```



```
In [19]: phone1 = phone('Anish')
         phone2 = phone('TQ')

In [20]: phone1.install_app('tiktok', 1000)
         phone1.install_app('IG', 300)
         phone2.install_app('pokego', 500)
         phone2.install_app('youtube', 100)

In [ ]:  phone.average_app_size()
```

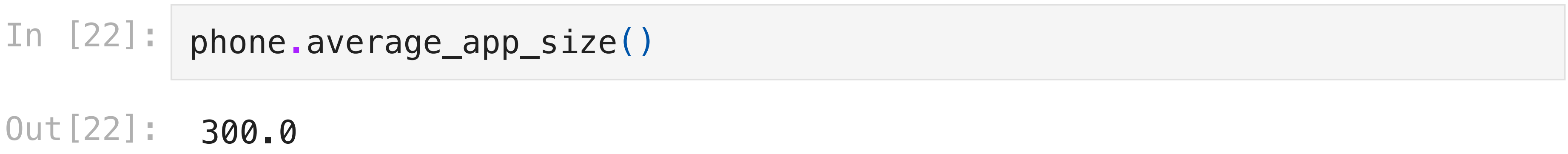
A. 0

B. 200

C. 475

D. 300

Checkpoint Solution



```
phone.average_app_size()
```

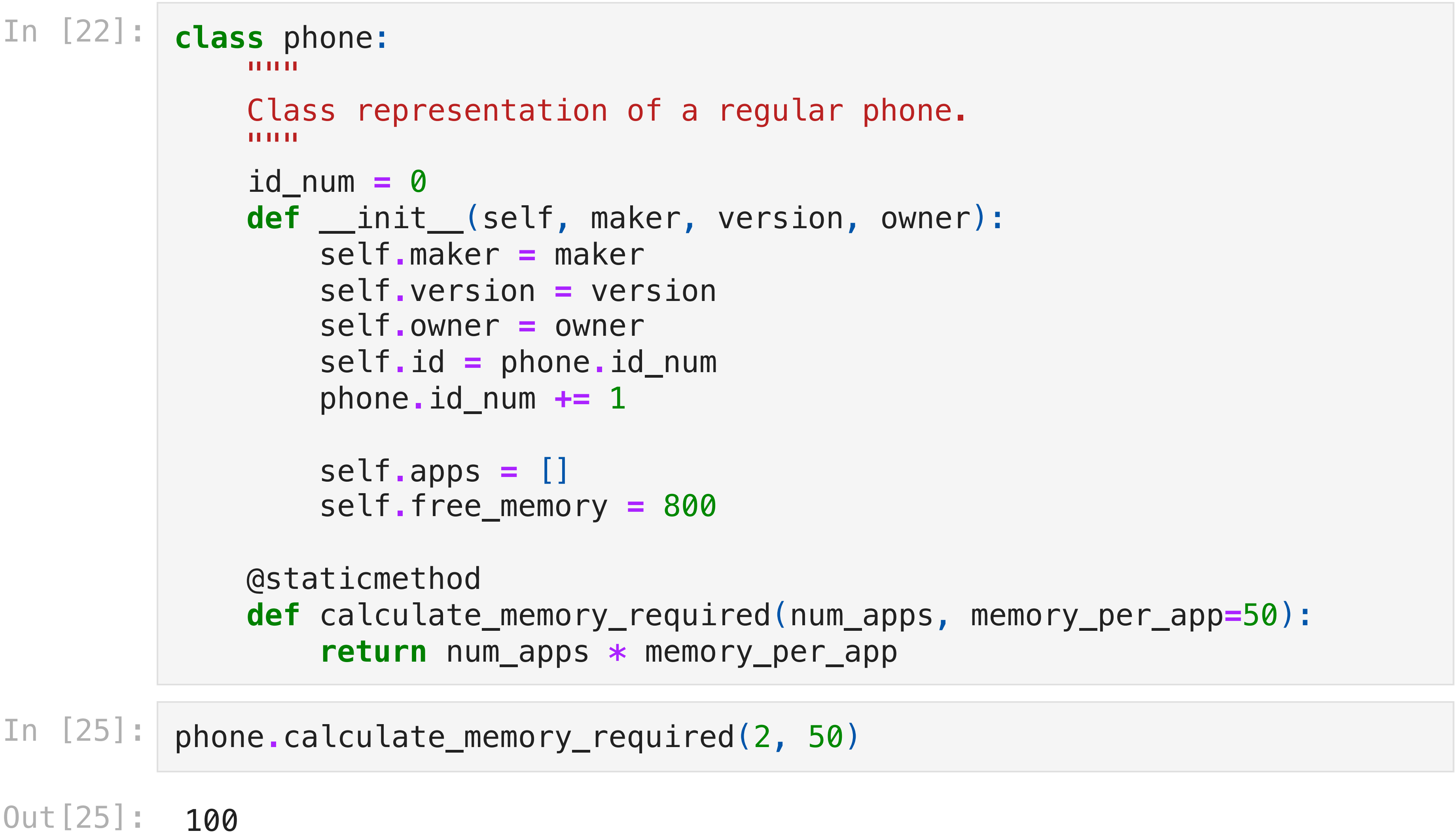
```
300.0
```

@staticmethod

The `@staticmethod` decorator transforms a method into a static method. Static methods, unlike class methods or instance methods, do not require a reference to the instance (`self`) or class (`cls`). This makes them behave more like plain functions that happen to reside within the class.

- **No Automatic Arguments:** `@staticmethod` functions do not automatically receive the `cls` or `self` arguments. They behave like regular functions but belong to the class's scope.

- **Use Cases:** Static methods are used when some processing is related to the class but does not require the class or its instances to perform its task. They can be called on the class itself or on instances of the class and are usually utility or helper functions.



In [22]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    id_num = 0
    def __init__(self, maker, version, owner):
        self.maker = maker
        self.version = version
        self.owner = owner
        self.id = phone.id_num
        phone.id_num += 1

        self.apps = []
        self.free_memory = 800

    @staticmethod
    def calculate_memory_required(num_apps, memory_per_app=50):
        return num_apps * memory_per_app
```

In [25]:

```
phone.calculate_memory_required(2, 50)
```

Out[25]:

```
100
```

Checkpoint

Given the following code, what is the result of the final expression?

In [24]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    id_num = 0
    def __init__(self, owner):
        self.owner = owner
        self.id = phone.id_num
        phone.id_num += 1
        self.apps = []
        self.free_memory = 800

    @staticmethod
    def validate_phone_number(number):
        if len(number) != 12:
            return False
        for i, char in enumerate(number):
            if i in [3, 7]:
                if char != '-':
                    return False
            elif not char.isdigit():
                return False
        return True
```

In [29]:

```
num1 = '123-456-7890'
num2 = '1234567890'
myphone = phone('ben')
print(myphone.validate_phone_number(num1))
print(phone.validate_phone_number(num2))
```

A. True, True

B. False, False

C. True, False

D. Error, Error

E. Error, False

Checkpoint Solution

In [31]:

```
print(myphone.validate_phone_number(num1))
print(phone.validate_phone_number(num2))
```

```
True
False
```

All Together

In [33]:

```
class phone:
    """
    Class representation of a regular phone.
    """
    id_num = 0
    def __init__(self, maker, version, owner):
        self.maker = maker
        self.version = version
        self.owner = owner
        self.id = phone.id_num
        phone.id_num+=1

        self.apps = []
        self.free_memory = 800

    @staticmethod
    def calculate_memory_required(num_apps, memory_per_app=50):
        return num_apps * memory_per_app

    @classmethod
    def get_idnum(cls):
        return cls.id_num

    def install_apps(self, *apps, memory=50):
        mem_req = phone.calculate_memory_required(len(apps), memory)
        if mem_req <= self.free_memory:
            self.free_memory -= mem_req
            self.apps+=apps
            return True
        return False
```

In [82]:

```
iphone = phone('Apple', 'X', 'Nikki')
print(iphone.get_idnum())
print(iphone.install_apps('genshin','honkai','league',memory=100))
print(iphone.apps)
```

```
1
True
['genshin', 'honkai', 'league']
```

Inheritance

- classes can have a "parent-child" relationship
- child class(es) "inherit" methods from their parent class
- use "is-a" paradigm to distinguish; given a parent class phone and a child class pearphone:
    - pearphones are phones (child is a parent)
    - but not all phones are pearphones (parents are not children)

In [83]:

```
class pearphone(phone):
    pass
    # since nothing is changed in my pearphone class,
    # it is currently just another name for phone class
```

In [84]:

```
pear = pearphone('pear', '9', 'Tim Raw')
print(isinstance(pear, pearphone))
print(isinstance(pear, phone))
print(isinstance(iphone, pearphone))
print(isinstance(iphone, phone))
```

```
True
True
False
True
```

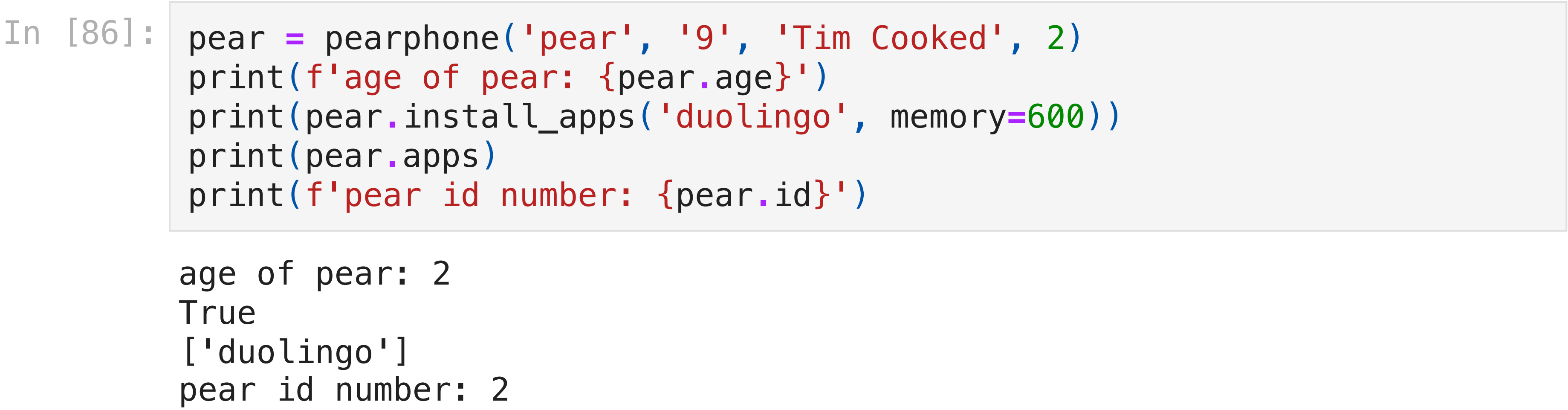
Inheritance (cont.)

- Though not required, child classes can **overwrite** methods of their parent class
- if certain things are to be retained, super() is a useful function to use

note: super refers to the direct parent of a class

In [85]:

```
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        self.maker = maker
        self.version = version
        self.age = age
        self.id = phone.id_num
        phone.id_num+=1
        # notice how I can still use ID from phone!

        self.apps = []
        self.free_memory = 800
```



In [86]:

```python
pear = pearphone('pear', '9', 'Tim Cooked', 2)
print(f'age of pear: {pear.age}')
print(pear.install_apps('duolingo', memory=600))
print(pear.apps)
print(f'pear id number: {pear.id}')
```

```
age of pear: 2
True
['duolingo']
pear id number: 2
```

Checkpoint

Given the following code, what is the result of the final expression?

In [34]:

```
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        self.maker = maker
        self.version = version
        self.age = age
        self.id = phone.id_num
        phone.id_num+=1
        self.apps = []
        self.free_memory = 800
```

In [ ]:

```
my_phone = pearphone('pear', 'Y', 'ben', 12)
my_phone.install_apps('chrome', 'youtube', 'instagram', memory=100)
my_phone.apps
```

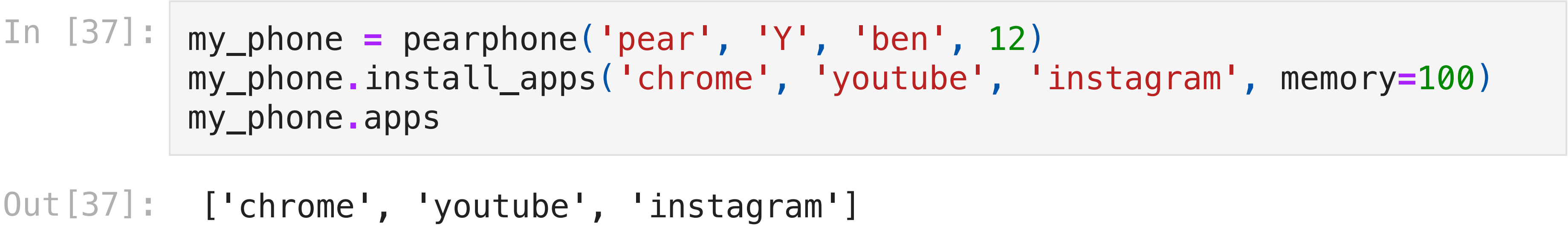
A. ['chrome', 'youtube', 'instagram']

B. []

C. ['youtube', 'instagram']

D. Error

Checkpoint Solution

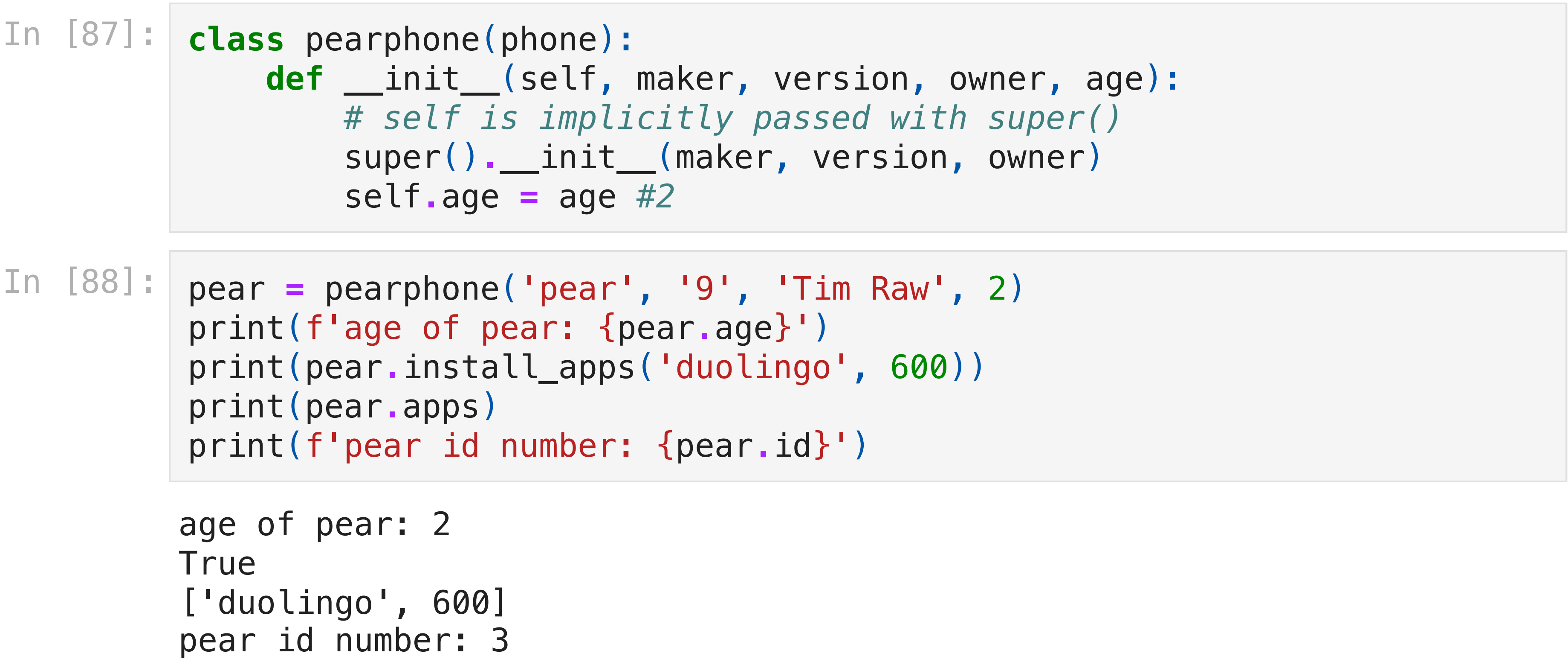


```
In [37]: my_phone = pearphone('pear', 'Y', 'ben', 12)
         my_phone.install_apps('chrome', 'youtube', 'instagram', memory=100)
         my_phone.apps

Out[37]: ['chrome', 'youtube', 'instagram']
```

super()

In pearphone, even though we want a different constructor, you can see that we actually reused a lot of the same code as the constructor for its parent class, phone. To avoid this, we can directly access the parent method using super(), and then add parameters we need.

explore super() more on your own :)



In [87]:

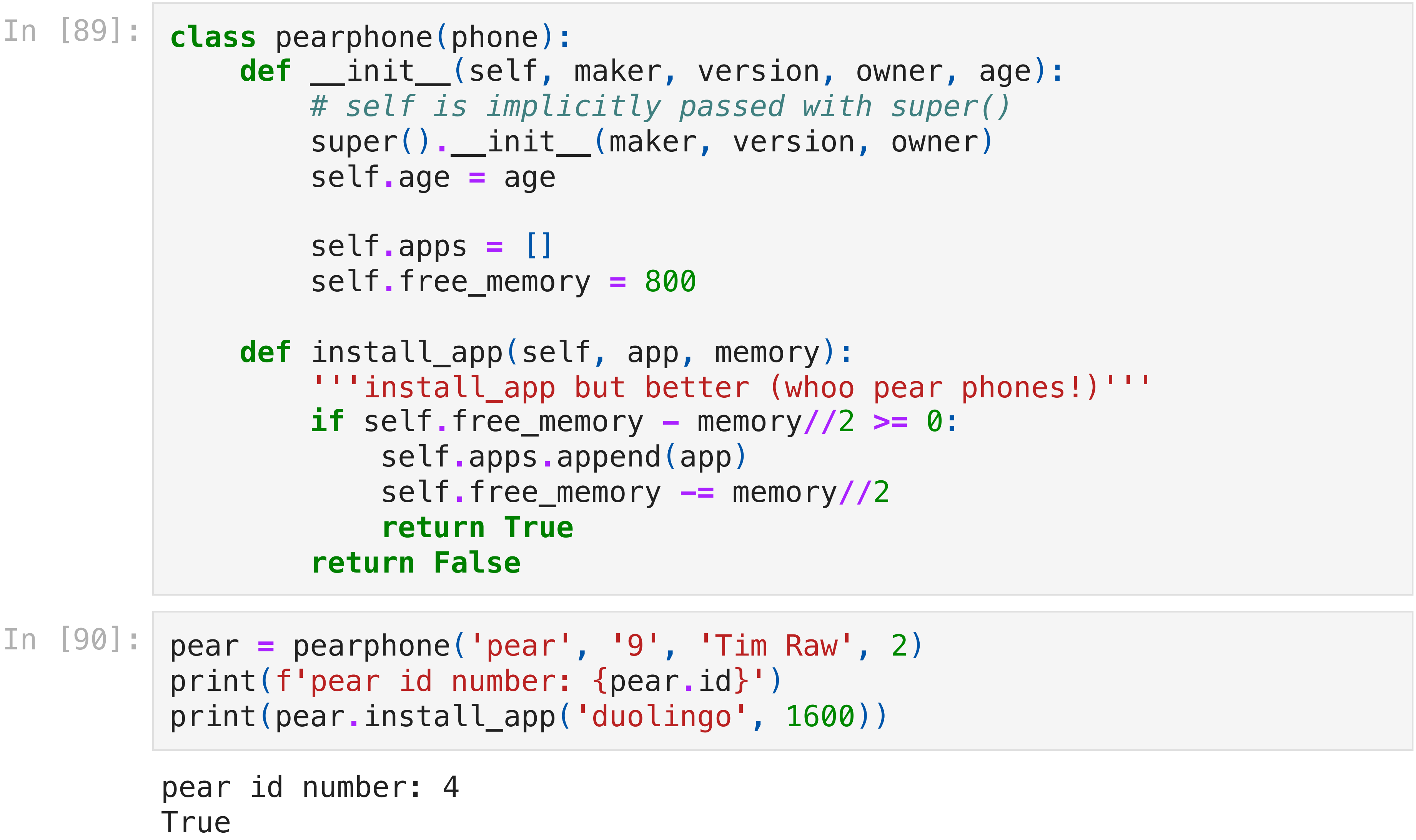
```python
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        # self is implicitly passed with super()
        super().__init__(maker, version, owner)
        self.age = age #2
```

In [88]:

```python
pear = pearphone('pear', '9', 'Tim Raw', 2)
print(f'age of pear: {pear.age}')
print(pear.install_apps('duolingo', 600))
print(pear.apps)
print(f'pear id number: {pear.id}')
```

```
age of pear: 2
True
['duolingo', 600]
pear id number: 3
```

Overwriting inherited methods



In [89]:

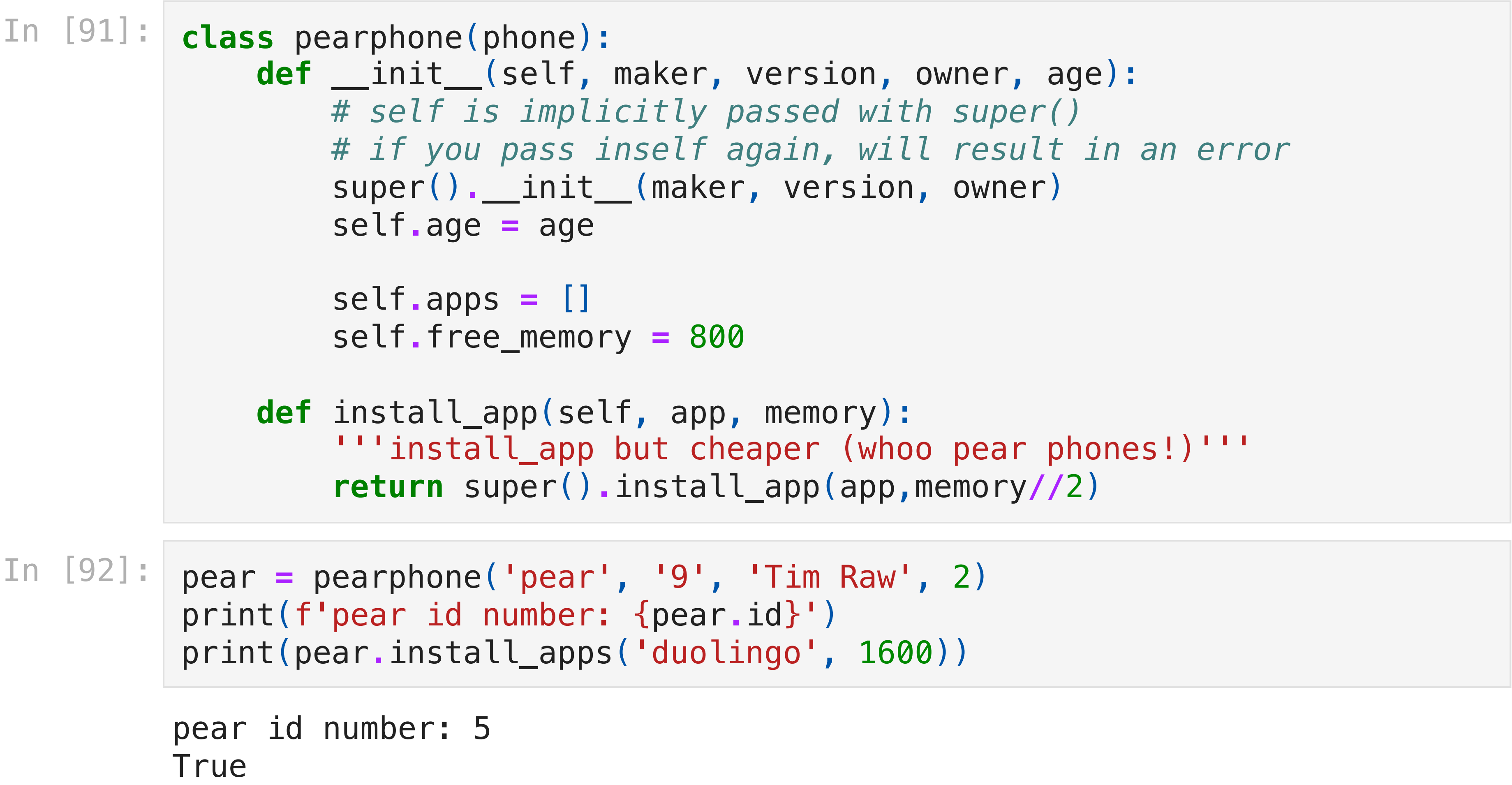
```python
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        # self is implicitly passed with super()
        super().__init__(maker, version, owner)
        self.age = age

        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        '''install_app but better (whoo pear phones!)'''
        if self.free_memory - memory//2 >= 0:
            self.apps.append(app)
            self.free_memory -= memory//2
            return True
        return False
```

In [90]:

```python
pear = pearphone('pear', '9', 'Tim Raw', 2)
print(f'pear id number: {pear.id}')
print(pear.install_app('duolingo', 1600))
```

```
pear id number: 4
True
```

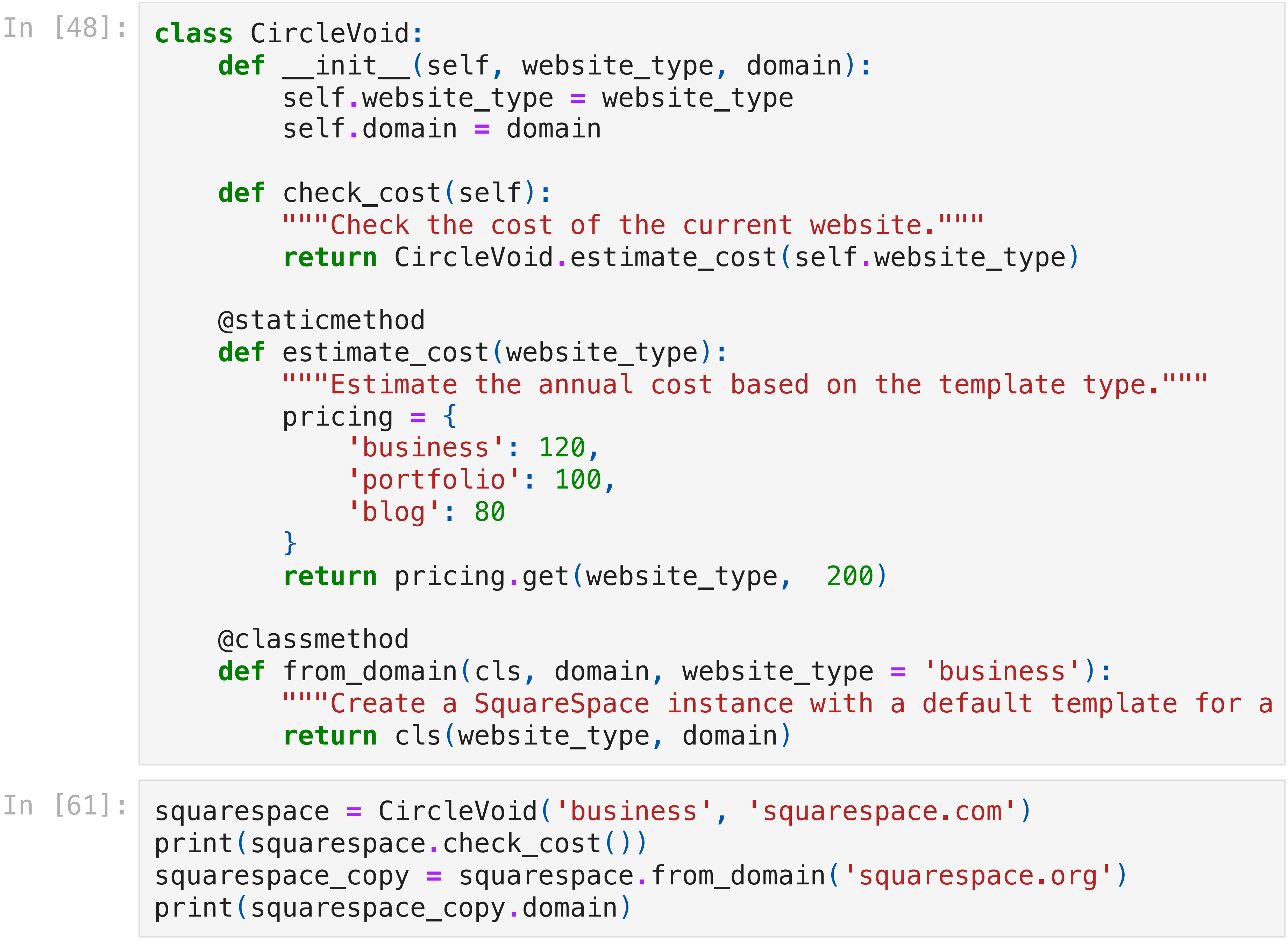
Using super...



```
In [91]:
class pearphone(phone):
    def __init__(self, maker, version, owner, age):
        # self is implicitly passed with super()
        # if you pass inself again, will result in an error
        super().__init__(maker, version, owner)
        self.age = age

        self.apps = []
        self.free_memory = 800

    def install_app(self, app, memory):
        '''install_app but cheaper (whoo pear phones!)'''
        return super().install_app(app,memory//2)
```

```
In [92]:
pear = pearphone('pear', '9', 'Tim Raw', 2)
print(f'pear id number: {pear.id}')
print(pear.install_apps('duolingo', 1600))
```

```
pear id number: 5
True
```

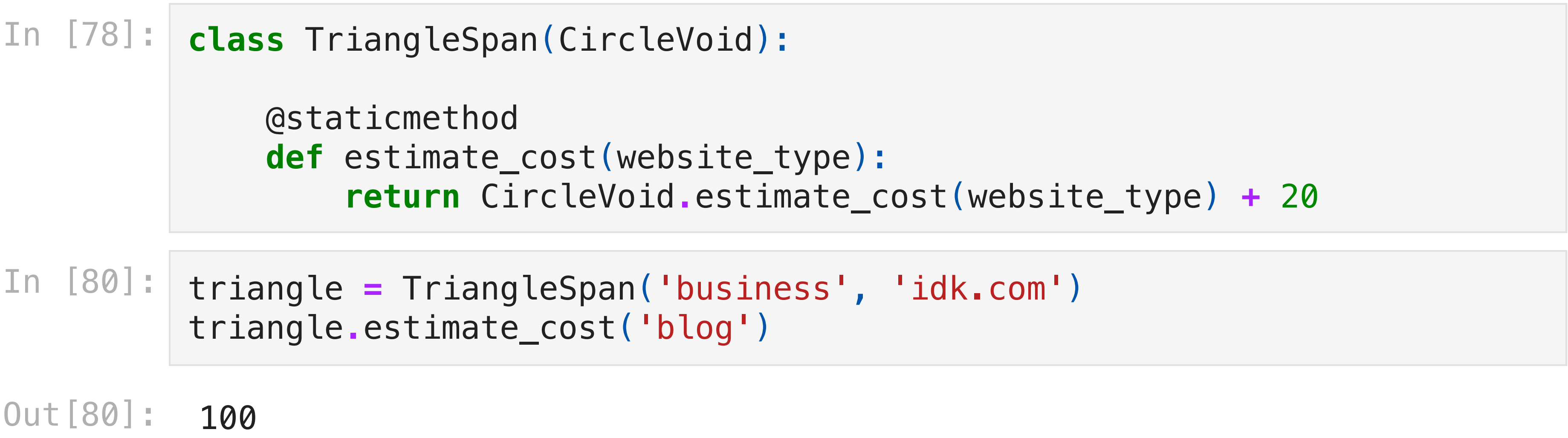


In [48]:

```python
class CircleVoid:
    def __init__(self, website_type, domain):
        self.website_type = website_type
        self.domain = domain

    def check_cost(self):
        """Check the cost of the current website."""
        return CircleVoid.estimate_cost(self.website_type)

    @staticmethod
    def estimate_cost(website_type):
        """Estimate the annual cost based on the template type."""
        pricing = {
            'business': 120,
            'portfolio': 100,
            'blog': 80
        }
        return pricing.get(website_type,  200)

    @classmethod
    def from_domain(cls, domain, website_type = 'business'):
        """Create a SquareSpace instance with a default template for a
        return cls(website_type, domain)
```

In [61]:

```python
squarespace = CircleVoid('business', 'squarespace.com')
print(squarespace.check_cost())
squarespace_copy = squarespace.from_domain('squarespace.org')
print(squarespace_copy.domain)
```



In [78]:

```python
class TriangleSpan(CircleVoid):

    @staticmethod
    def estimate_cost(website_type):
        return CircleVoid.estimate_cost(website_type) + 20
```

In [80]:

```python
triangle = TriangleSpan('business', 'idk.com')
triangle.estimate_cost('blog')
```

Out[80]:

```
100
```

Thanks for coming!