Practical 5: Objects

From Functions to Classes

Table of contents

1 One Last Time Playing with DoLs 2

2 Appending a Column 2

3 For Loops Without For Loops 3

4 'Functionalising' 4

5 Decorating! 7

6 Switching to Object-Oriented Code 8

7 Improving the Documentation 11

8 Packaging It Up 12

9 Classes and Inheritance 13

This is a very challenging notebook because it takes you *both* through the process of building a function incrementally *and* through a 'simple' example of how Python classes actually work. You will need to understand these two very different elements in order to make the most of the remaining 6 weeks of term, because we both improve our code incrementally *and* make use of objects and their inheritances extensively. You also get an extra chance to revisit the differences between LoLs and DoLs because you will undoubtedly encounter and make use of these data structures even *after* you become a skillfull Python programmer.

> ⚠ Warning
>
> This is a very challenging practical and you should do your best to ensure that you actually understand what you have done and why.

> 💡 Group Sign-Up
>
> You should now make it a priority Sign Up!

1 One Last Time Playing with DoLs

The preceding practical is hard, so I want to provide *another* chance for the concepts to bed in before we use them in an *object-oriented way through Pandas*. Yes, Week 5 will show how we combine concepts covered over the preceding two weeks in *practice* to begin to 'do data science'.

First, remember the finding from last week: if we don't really care about column order, then a dictionary of lists is a nice way to handle data. And why should we care about column order? With our CSV file we saw what a pain it was to fix things when even a tiny thing like the layout of the columns changed.

But if, instead, we could just reference the 'Description' column in the data set then it doesn't matter where that column actually is *and* we would know that all the descriptions would be *text*, while all the populations or prices would be *numbers*. Why is that?

> **Connections**
>
> This task briefly recaps the final part of the previous practical and builds on the DOLs to Data and Functions lectures.

```
myData = {
    'id'          : [0, 1, 2, 3, 4],
    'Name'        : ['Gorgeous 2 bed flat w easy access to Earlsfiel...', 'Welcome to
    'Longitude'   : [-0.189030, -0.093411, -0.022240, -0.078328, -0.029900],
    'Latitude'    : [51.442430, 51.593397, 51.499260, 51.525488, 51.514680],
    'Bedrooms'    : [2, 1, 2, 1, 3],
}
```

To print out a list of every listing in the data set:

```
print(", ".join(myData['Name']))
```

To find out if `Out in Dalston` is included in the list of data:

```
if 'Out in Dalston' in myData['Name']:
    print("Found 'Out in Dalston' in the data set!")
else:
    print("Didn't find 'Out in Dalston' in the data set.")
```

See how even basic questions like "Is X in our data set?" are now easy (and quick) to answer? We no longer need to loop over the entire data set in order to find one data point. In addition, we know that everything in the 'Name' column will be a string, and that everything in the 'Longitude' column is a float, while the 'Bedrooms' column contains integers. So that's made life easier already. But let's test this out and see how it works.

2 Appending a Column

2.1 Calculate Mean

Let's start by calculating the sample mean (use Google: `Python numpy mean...`):

Question

```
import numpy as ??
# Use numpy function to calculate mean
mean = ??.??(myData['Bedrooms'])
print(f"The mean. number of bedrooms is {mean:,.1f}.")
```

2.2 Calculate Standard Deviation

Difficulty level: Low-ish.

Now let's do the standard deviation:

Question

```
import numpy as np
# Use a numpy function to calculate the standard deviation
std  = np.??(??)
print(f"The standard deviation of bedrooms is {std:,.2f}.")
```

So the `numpy` package gives us a way to calculate the mean and standard deviation *quickly* and without having to reinvent the wheel. The other potentially new thing here is `{std:,.2f}`. This is about string formatting and the main thing to recognise is that this means 'format this float with commas separating the thousands/millions and 2 digits to the right'. The link I've provided uses the slightly older approach of `<str>.format()` but the formatting approach is the same.

3 For Loops Without For Loops

Difficulty level: Medium.

Now we're going to see something called a **List Comprehension**.

In Python you will see code like this a lot: `[x for x in list]`. This syntax is known as a 'list comprehension' and is basically a `for` loop on one line with the output being assigned to a list. So we can apply an operation (converting to a string, subtracting a value, etc.) to every item in a list without writing out a full for loop.

Here's a quick example just to show you what's going on:

```
demo = range(0,10) # <- a *range* of numbers between 0 and 9 (stop at 10)
print([x**2 for x in demo]) # square every element of demo
```

Now let's apply this to our problem. We calculated the the mean and standard deviation above, so now we want to apply the z-score formula to every element of the Population list… Remember that the format for the z-score (when dealing with a sample) is:

$$z = \frac{x - \bar{x}}{s}$$

And the population standard deviation (by which I mean, if you are dealing with *all* the data, and not a subsample as we are here) is:

$$z = \frac{x - \mu}{\sigma}$$

Question

```
rs = [(x - ??)/?? for x in myData['Bedrooms']] # rs == result set
print([f"{x:.3f}" for x in rs])
```

3.1 Appending

Difficulty level: trivial

And now let's add it to the data set:

```
myData['Std. Bedrooms'] = rs
print(myData['Std. Bedrooms'])
```

And just to show how everything is in a single data structure:

```
for c in myData['Name']:
    idx = myData['Name'].index(c)
    print(f"Listing {c} has {myData['Bedrooms'][idx]:,} bedrooms and standardised sc
```

4 'Functionalising'

Let's start trying to pull what we've learned over the past two weeks together by creating a function that will download a file from a URL (checking if it has already *been* downloaded to save bandwidth).

To be honest, there's not going to be much about writing our *own* objects here, but we will be making use of them and, conceptually, an understanding of objects and classes is going to be super-useful for understanding what we're doing in the remainder of the term!

4.1 Downloading from a URL

Let's focus on the first part *first* because that's the precondition for everything else. If we can get the 'download a file from a URL' working then the rest will gradually fall into place through *iterative* improvments!

4.1.1 Finding an Existing Answer

> Difficulty level: Low

First, let's be sensibly lazy–we've already written code to read a file from the Internet and save it locally. We're going to start from this point.

```
from urllib.request import URLError
from urllib.request import urlopen

try:
    url = 'https://orca.casa.ucl.ac.uk/~jreades/data/Listings.csv'
    response = urlopen(url)
    raw  = response.read()
    data = raw.decode('utf-8')
    with open('Listings.csv', 'w') as f:
        f.writelines(data)
except URLError as e:
    print(f"Unable to download data: {e}")
```

4.1.2 Getting Organised

> Difficulty level: Low

Let's take the code above and modify it so that it is:

1. A function that takes two arguments: a URL and a destination filename.
2. A function that checks if a file exists already before downloading it again.

You will find that the `os` module helps here because of the `path` function. And you will need to Google how to test if a file exists. I would normally select a StackOverflow link in the results list over anything else because there will normally be an *explanation* included of why a particular answer is a 'good one'. I also look at which answers got the most votes (not always the same as the one that was the 'accepted answer'). In this particular case, I also found this answer useful.

I would start by setting my inputs:

```
import os
url = "https://orca.casa.ucl.ac.uk/~jreades/data/Listings.csv"
out = os.path.join('data','Listings.csv') # Print `out` if you aren't sure what this
```

4.1.3 Sketching the Function

> Difficulty level: Low, if you've watched the videos…

Then I would sketch out how my function will work using comments. And the simplest thing to start with is checking whether the file has already been downloaded:

Question

```
from urllib.request import urlopen

def get_url(src, dest):

    # Check if dest exists -- if it does
    # then we can skip downloading the file,
    # otherwise we have to download it!
    if os.path.isfile(??):
        print(f"{dest} found!")
    else:
        print(f"{dest} *not* found!")

get_url(url, out)
```

4.1.4 Fleshing Out the Function

> Difficulty level: Medium
>
> If you really explore what's going on in the function rather than just running it and moving on.

I would then flesh out the code so that it downloads the file if it isn't found and then, either way, returns the *local* file path for our CSV reader to extract:

```
def get_url(src, dest):

    # Check if dest does *not* exist -- that
    # would mean we had to download it!
    if os.path.isfile(dest):
        print(f"{dest} found locally!")
    else:
        print(f"{dest} not found, downloading!")

        # Get the data using the urlopen function
        response = urlopen(src)
        filedata = response.read().decode('utf-8')

        # Extract the part of the dest(ination) that is *not*
```

```
        # the actual filename--have a look at how
        # os.path.split works using `help(os.path.split)`
        path = list(os.path.split(dest)[:-1])

        # Create any missing directories in dest(ination) path
        # -- os.path.join is the reverse of split (as you saw above)
        # but it doesn't work with lists... so I had to google how
        # to use the 'splat' operator! os.makedirs creates missing
        # directories in a path automatically.
        if len(path) >= 1 and path[0] != '':
            os.makedirs(os.path.join(*path), exist_ok=True)

        with open(dest, 'w') as f:
            f.write(filedata)

        print(f"Data written to {dest}!")

    return dest

# Using the `return contents` line we make it easy to
# see what our function is up to.
src = get_url(url, out)
```

5 Decorating!

Let's now look into simplifying this code using a dectorator! Our function has become a bit unwieldy and we want to look at how we can simplify that.

The 'obvious' (i.e. not obvious) way to do this is to implement the check for a local copy as a decorator on the downloading function. So we have a function that downloads, and a decorator function that checks if the download should even be triggered.

```
from functools import wraps
def check_cache(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        src  = args[0]
        dest = args[1]
        if os.path.isfile(dest):
            print(f"{dest} found locally!")
            return(dest)
        else:
            print(f"{dest} not found, downloading!")
            return(f(src, dest))
    return wrapper

@check_cache
def get_url(src, dest):
    # Get the data using the urlopen function
```

```
    response = urlopen(src)
    filedata = response.read().decode('utf-8')

    # Extract the part of the dest(ination) that is *not*
    # the actual filename--have a look at how
    # os.path.split works using `help(os.path.split)`
    path = list(os.path.split(dest)[:-1])

    # Create any missing directories in dest(ination) path
    # -- os.path.join is the reverse of split (as you saw above)
    # but it doesn't work with lists... so I had to google how
    # to use the 'splat' operator! os.makedirs creates missing
    # directories in a path automatically.
    if len(path) >= 1 and path[0] != '':
        os.makedirs(os.path.join(*path), exist_ok=True)

    with open(dest, 'w') as f:
        f.write(filedata)

    print(f"Data written to {dest}!")

    return dest

# Using the `return contents` line we make it easy to
# see what our function is up to.
src = get_url(url, out)
print(f"{src} is a {type(src)}.")
```

```
data/Listings.csv found locally!
data/Listings.csv is a <class 'str'>.
```

I'm not going to pretend that's the *best* use of a decorator, but it *does* neatly separate the downloading function from the caching function. In fact, there is already a cache decorator and some of these have unlimited capacity; however, they are intended to run in a 'live' context, so you'd still need to download the file again any time you start a new notebook or restart Podman. *This* caching function saves the actual data locally to `dest`.

> ⚠ Stop!
>
> It really would be a good idea to put in the effort to make sense of how this function works. There is a lot going on here and understanding how this function works will help you to understand how to code. You should notice that we don't try to check if the data file contains any useful data! So if you download or create an empty file while testing, you won't necessarily get an error until you try to turn it into data afterwards!

6 Switching to Object-Oriented Code

While the `os` package is still widely used, not long ago the people who contributed to Python decided that it would be better to be consistent and that paths (file and directory locations) on

computers should use an object-oriented approach too. So all Python code will gradually shift to using `pathlib` instead and this is a good place to demonstrate why going object-oriented is a *choice* and not a *requirement*.

6.1 Pathlib

Pathlib is intended to work around the fact that different Operating Systems represent paths on file systems in different ways. Using the `os` library there is quite a lot of 'faff' involved in managing this: whereas Unix, Linux, and the macOS (which is BSD Unix under the hood) use forward slashes in a path (e.g. `/Users/<username>/Documents/...`)[1], Windows usess backslashes (e.g. `C:\Users\<username>\My Documents\...`). The differences don't stop there, but that's enough to see how life can get complicated; in `os` this is managed by the bundle of functions and constants under `os.path` and you can then use `os.path.sep.join([<a list representing a path on the computer>])`.

Pathlib does away with all of that. Remember that we've defined an object as a bundle of data and functions that act on the data. So a `Path` object is *both* data about a location on a computer *and* a set of functions that can manipulate that data. Let's see what that means in practice:

```
from pathlib import Path

here = Path('.')
print("Using a directory: ")
print(f"\tHere: {here}")
print(f"\tParents: {here.parents}")
print(f"\tURI: {here.absolute().as_uri()}")
print(f"\tIs directory: {here.is_dir()}")
print(f"\tIs file: {here.is_file()}")
print(f"\tParts: {here.resolve().parts}")
print(f"\tDirectory contents: \n\t\t{'\n\t\t'.join([str(x) for x in list(here.glob('

print()

try:
    fn = here / 'Practical-05-Objects.ipynb' # Try changing this!
    print("Using a file:")
    print(f"\tResolved path: {fn.resolve()}")
    print(f"\tExists: {fn.exists()}")
    print(f"\tIs file: {fn.is_file()}")
    print(f"\tFile size (bytes): {fn.stat().st_size:,} bytes")
    print(f"\tOwner: {fn.owner()}")
except FileNotFoundError:
    print("File not found!")
```

Hopefully you can now see how the object-oriented approach of `pathlib` allows to write neater, more readable code than the older `os` module? We no longer have to keep passing in a list or string representing a path on a specific operating system, we can just pass around Path objects and let Python take care of turning them into concrete paths on a computer when we need to 'act' on the Path in some way (check that it exists, create it, interrogate it, etc.).

[1] Slight exception to this: the macOS *also* recognises `:` as a path separator

> **Updating…**
>
> In fact, I'm still updating the practicals to make full use of `pathlib` so you may find places where the older `os` approach is still in use. If you spot one of these why not submit an Issue or even fix it in a copy of the FSDS repo and then submit a Pull Request on GitHub to correct the code?

6.2 Updating the Function

So let's update the function to move from `os` to `pathlib` and see how things change…

```
from functools import wraps
from pathlib import Path
def check_cache(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        src  = Path(args[0])
        dest = Path(args[1])
        if dest.exists() and dest.is_file():
            print(f"{dest} found locally!")
            return(dest)
        else:
            print(f"{dest} not found, downloading!")
            return(f(src, dest))
    return wrapper

@check_cache
def get_url(src, dest):
    # Get the data using the urlopen function
    response = urlopen(src)
    filedata = response.read().decode('utf-8')

    # Create any missing directories in dest(ination) path
    # -- os.path.join is the reverse of split (as you saw above)
    # but it doesn't work with lists... so I had to google how
    # to use the 'splat' operator! os.makedirs creates missing
    # directories in a path automatically.
    dest.parent.mkdir(parents=True, exist_ok=True)

    with dest.open(mode='w') as f:
        f.write(filedata)

    print(f"Data written to {dest}!")

    return dest

# Using the `return contents` line we make it easy to
# see what our function is up to.
src = get_url(url, out)
print(f"{src} is a {type(src)}.") # <- Note this change!
```

7 Improving the Documentation

> Difficulty: Low

To help people make use of the newly-improved function, we could useful add two types of information to the function:

1. Use the 'docstring' support offered by Python so that `help(...)` gives useful output.
2. Provided hints to Python about the expected input and output data types.

```
from functools import wraps
from pathlib import Path
def check_cache(f) -> Path:
    """
    Checks if a file exists locally before forwarding to a 'getter'
    function if it's not found.

    :param src: any remote file path
    :type src: str
    :param dst: any local file path
    :type dst: str
    :returns: the local file path
    :rtype: pathlib.Path
    """
    @wraps(f)
    def wrapper(*args, **kwargs):
        src  = Path(args[0])
        dest = Path(args[1])
        if dest.exists() and dest.is_file():
            print(f"{dest} found locally!")
            return(dest)
        else:
            print(f"{dest} not found, downloading!")
            return(f(src, dest))
    return wrapper

@check_cache
def get_url(src:Path, dst:Path) -> Path:
    """
    Reads a remote file (src) and writes it to a local file (dst),
    returning a Path object that is the location of the saved data.

    :param src: any remote file path
    :type src: pathlib.Path
    :param dst: any local file path
    :type dst: pathlib.Path
    :returns: the local file path
    :rtype: pathlib.Path
    """
```

```
    # Get the data using the urlopen function
    response = urlopen(src)
    filedata = response.read().decode('utf-8')

    # Create any missing directories in dest(ination) path
    # -- os.path.join is the reverse of split (as you saw above)
    # but it doesn't work with lists... so I had to google how
    # to use the 'splat' operator! os.makedirs creates missing
    # directories in a path automatically.
    dst.parent.mkdir(parents=True, exist_ok=True)

    with dst.open(mode='w') as f:
        f.write(filedata)

    print(f"Data written to {dst}!")

    return dst
```

```
help(check_cache)
help(get_url)
```

8 Packaging It Up

> **Optional Content**
>
> This content is optional but will help you to understand how to move functions out of the core namespace and into groups of related tools.

We're not going to tackle this now, but it's important that you understand how what we've done connects to what we're *about* to do, and the concept of a package is the bridge. We've already covered this in the pre-recorded lectures, but if you want to actually *try* to create your own package, the simplest way to do this is to:

1. Copy the two functions above (`check_cache` and `get_url`) into a new file called, for instance, `utils.py`.
2. Make sure you delete this function from the current 'namespace' (`del(get_url)`) by which I mean that the running `help(get_url)` should give you an error!.
3. Try importing the function from the file: `from utils import get_url` and run the `help(get_url)` code again.

Assuming that you've done everything correctly, we've now brought in code from another file without having to write it into our main Python script file. In Python, many of the most complex libraries are spread across the equivalent of *many* `utils.py` files, but on top of *that* when we import and run them they are also creating objects from classes defined in those files.

9 Classes and Inheritance

> **Optional Content**
>
> This content is optional but will help you to understand how object-oriented programming works and how you can develop your own classes.

Before you get stuck into this section, bear in mind that we're *not* saying you should use Object-Oriented approaches to every programming problem you encounter. OO carries quite a high cognitive and computation overhead compared to just packaging up useful functions that return outputs you manipulate directly.

What we're doing is trying to show *how* classes are built and how they work using a conceptual framework that builds on data you're already broadly familiar with. You can find a decent overview of programming 'styles' easily enough online but really understanding when each is appropriate is a much bigger proposition. You will tend to find OO used with *applications* in which different bits need to pass information back and forth in structured ways (e.g. who is the user and what privileges do they have? what data can go into this chart? etc.)

> **Difficulty:** .

In Python, many of the most complex libraries are spread across the equivalent of *many* `utils.py` files, but on top of *that* when we import and run them they are also creating objects from classes defined in those files. What we now want to do is use a fairly simple example using a basic representation of Airbhb listings that allow us to explore how classes work through inheritance from parents and can extend of overwrite the functionality provided by the parent class. We'll need this understanding in order to grasp how Pandas and GeoPandas work specifically, but also how Python works more generally.

> **Connections**
>
> This will draw on what you've learned in the lectures about Methods, Classes, and Design. You will also find the Code Camp Classes session useful.

We want to create a set of classes to 'help' Airbnb manage bookings – these will necessarily be highly simplified and are intended more as a teaching resource than a demonstration of outstanding Object-Oriented Design; however, it's my hope that you'll see how classes and objects can support programming and application development.

We're also going to make use of a few features of Python:

- You can access the class name of an instance using: `self.__class__.__name__`. And here's one key point: `self` refers to the instance (to *this* particular shape that I've created), not to the class in general (to *all* objects of the same class)… we'll see why this matters.
- You can raise your own exceptions easily if you don't want to implement a particular method yet. This is giving you control over how your code behaves when something goes 'wrong' – as we've covered elsewhere sometimes an error is 'expected' and we want to handle the *exception*, other times it is 'unexpected' and we're going to let Python fail so that the user knows something is seriously wrong.
- You can have an 'abstract' base class that does nothing except provide a template for the 'real' classes so that different types of listings can be used interchangeably. This is quite an advanced feature, but it gives our script a lot more flexibility: we don't need to worry

about whether we're working with a flat, house, or houseboat because they are defined in a way that allows this flexibility.

9.1 Abstract Base Class

This class appears to do very little, but there are two things to notice:

1. It provides a constructor (`__init__`) that sets the `listing_type` to the name of the class automatically (so a `flat` object has `shape_type='Flat'`) and it stores the critical accommodation details.
2. It provides methods (which only raise exceptions) that will allow all listings to be used interchangeably in the right context (e.g. determining availability).

```
# We first need to install a library for managing
# availability in a highly optimised way.
try:
    from bitarray import bitarray
except:
    ! pip install bitarray
    from bitarray import bitarray
```

```
from bitarray import bitarray

# This automatically generates methods and
# other useful features for child classes
from dataclasses import dataclass, field

@dataclass
class Listing(object): # Inherit from base class

    listing_name: str = field(default_factory=str)
    listing_price: float = field(default_factory=float)
    listing_bedrooms: int = field(default_factory=int)
    listing_availability: bitarray = field(default_factory=bitarray)

    def name(self) -> str:
        return self.listing_name

    def price(self) -> float:
        return self.listing_price

    def bedrooms(self) -> int:
        return self.listing_bedrooms

    def is_available(self, requested:bitarray) -> bitarray:
        return bitarray(self.listing_availability & requested).any()

    def type(self):
        return self.__class__.__name__
```

We can now create a new listing object (an *instance* of the Listing class) but we can't do much that is useful with it:

```
l = Listing('A listing', 25.00, 3, bitarray(14))

try:
    print(f"I am a {l.type()}")
    print(f"My name is {l.name()}")
except Exception as e:
    print(f"Error: {e}")
```

9.2 Flat

Implements a flat:

1. Adds 'Flat:' or 'Studio:' to all requests for the name.
2. Adds a 'floor' method to track what floor the flat is on.
3. Checks if the number of 'bedrooms' is 0 (in which case it's a 'Studio:') or more (in which case it's a 'Flat:').

Question

Can you work out the missing elements that will allow you to create a flat class? We'll want to know the floor and whether we're dealing with a Flat or a Studio by looking at the name of the listing.

```
# Flat class
class Flat(Listing): # Inherit from listing

    listing_floor = -99

    def __init__(self, name:str, price:float, bedrooms:int, floor:int, availability:
        ??.__init__(name, price, bedrooms, availability)
        self.listing_floor = ??

# If you've done everything correctly then
# you will no longer get an error here...
f = Flat('Room With a View', 73.00, 0, )

try:
    print(f"I am a {f.type()}")
    print(f"My name is {f.name()}")
except Exception as e:
    print(f"Error: {e}")
```

You should get:

- I am a `Flat`
- My name is `Studio: Room With a View`

If you were to create a *second* flat listing you can see how things change automatically based on the attributes:

```
f2 = Flat('Room With a Better View', 123.00, 2, 6, bitarray('10000001000000'))
print(f"I am a `{f2.type()}`")
print(f"My name is `{f2.name()}`")
```

- I am a `Flat`
- My name is `Flat: Room With a Better View`

9.3 Houseboat

Implements a houseboat listing:

1. Adds an ‘on_water’ boolean attribute.
2. Adds ‘Floating Palace:’ to the name if the houseboat is on water, or ‘Grounded!’ if it’s not.

Question

Can you work out the missing elements that will allow you to create a houseboat class?

```
# Houseboat class
class Houseboat(Listing): # Inherit from shape
    def __init__(self, ...):
        # Something...

    def on_water(self):
        # Something...

# If you've done everything correctly then
# you will no longer get an error here...
h = Houseboat(15)

try:
    print(f"I am a {h.type()}")
    print(f"My name is {h.name()}")
    if h.on_water():
        print("Where I expect a houseboat to be.")
    else:
        print("Something has gone very wrong!")
except Exception as e:
    print(f"Error: {e}")
```

9.4 Availability

The last thing we want to do is look at how (and why) we’ve recorded availability. Let’s pretend that this can only be measured 2 weeks in advance (i.e. 14 days). Let’s look at how the `bitarray` can help us here:

```
b1 = bitarray(14)
b2 = bitarray('11111111111111')
b3 = bitarray('01010101010101')

print(b1 & b2)
print(b2 & b3)
print(b1 | b2)
print(b1 | b3)
print(bitarray(b1 & b3).any())
print(bitarray(b1 | b3).any())
print(bitarray(b1 & b3).count())
print(bitarray(b1 | b3).count())
```

Does it now make sense why the `is_available` is set up the way it is?

9.5 Pulling It All Together

Let's finish with a demonstration of how we can use these classes in some kind of application:

Question

```
f1 = Flat('Room With a View', 73.00, 0, 5, bitarray('11000001100000'))
f2 = Flat('Room With a Better View', 123.00, 2, 6, bitarray('10000001000000'))
h1 = Houseboat('Bargemaster', 5.25, 1)
h2 = Houseboat('Sinking', 15.25, 1, bitarray('11111110000000'))

listings = [f1, f2, h1, h2]

# You are looking for availability on Mondays...
search = bitarray( ?? )

available = [x for x in listings if ??]

print("The following options are available: ")
for a in available:
    print(f"\t{a.name()} is available for £{a.price():,.2f}")
```

You should get that `Studio: Room With a View` and `Floating Palace: Sinking` are available.