Advanced exercise: Power generation

This exercise uses more advanced techniques, in particular it takes an object-oriented approach, and there will be less explanation of programming features. It is intended for people who are more confident with Python.

Introduction to Object-oriented programming

What are objects?

A lot of things you have used in Python are really objects. Objects contain variables used to store data (called attributes) and also contain functions that run code on themselves (called methods).

For example, we can import `datetime` and use it to create a `datetime` object called `current_datetime`.

```
from datetime import datetime
current_datetime = datetime.now()
```

This contains attributes like `year`, `month` and `day`. Note that these are variables stored within the `datetime` object, accessed using the `object.attribute` notation here.

```
print(f"Current year: {current_datetime.year}")
print(f"Current month: {current_datetime.month}")
print(f"Current day: {current_datetime.day}")
```

The `datetime` object contains various methods. For example, `strftime()` is used to output the `datetime` object as a string. Note that when we call an object's method we use the `object.method()` notation, so we don't call `strftime(current_datetime)` but instead do

```
formatted_date = current_datetime.strftime("%d/%m/%Y")
print(formatted_date)
```

Some methods might change the attributes. For example, `datetime` contains `replace()` which can be used to change some of its attributes. Here we change current_datetime to the same date and time next year.

```
next_year = current_datetime.replace(year=current_datetime.year+1)
```

Creating your own object

You create an object (also known as a class) using the command `class`.

Here we create a `class` called `Dog`. This is an abstract `Dog`, like you might have an abstract datetime or integer or string. We will not act on `Dog` directly, but create specific instances of it, like we created a specific `datetime` object above.

- If a class has an attribute that is shared by all instances, we can define this in the main body of the `class`, like here we define an attribute that all dogs have `legs = 4`. (Try not to think about this too much.)
- A class can have a method `__init__` which is performed when it is created. The values passed to methods in a class should start with `self`, i.e. telling the method to act on itself. Any other variables specified here are passed from the command that created the class. Here we are saying that to create a new `Dog` we must specify the `name` and `age`. These are stored as attributes specific to this `Dog`.
- As well as the attributes `legs`, `name` and `age`, we give our `Dog` a method, which is a function it can perform if requested. This is `bark` and it simply returns a totally convincing dog noise.

```
class Dog:
    legs = 4

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return "Woof! Woof!"
```

Now we have defined our `Dog`, we can create a couple of instances of this class, one we will store as `ted` and the other as `lil`.

```
ted = Dog("Ted", 5)
lil = Dog("Lil", 7)
```

We can access the attributes of our `Dog` objects:

```
print("{} is {} and has {} legs.".format(ted.name, ted.age, ted.legs))
print("{} is {} and has {} legs.".format(lil.name, lil.age, lil.legs))
```

We can also ask a `Dog` to run its method.

```
print(ted.bark())
```

Power generation simulation

Let's get on with modelling a power generation system.

The basic idea is that there is an electricity grid which handles an amount of power. We're going to create power generators, which add to this grid, and power users, who use the power. For simplicity, we're going to arrange this so that first our power generators generate their day's electricity, then the customers make their demands on the grid. We are interested in whether the generators produce enough energy to meet the needs of the customers.

First let's import the packages we are going to use.

```
from random import randint
import matplotlib.pyplot as plt
import math
```

We're going to store the amount of power in the grid as a global variable `grid`.

First the power generator class. This has an attribute `output` which stores the amount of energy it creates. It also contains a method `generate` which tells it to generate today's energy. This adds to the global variable `grid` an amount of energy produced. At first, we'll just have each generator produce a random amount of energy. We can try to make it more realistic later.

```
class power_generator:
    def __init__(self):
        self.output = 0

    def generate(self,day):
        global grid

        grid += randint(0,300)
```

Now we create our power users. These an initiated with a `level` of energy that they require each day. We'll assume this is constant for each power user. They have a method `consume` which tells them to draw the day's power from the grid, unless there isn't enough power, in which case they report a problem.

```
class power_user:
    def __init__(self,level):
        self.level = level

    def consume(self):
        global grid

        if grid >= self.level:
            grid -= self.level
            return True
        else:
            print("Not enough power")
            return False
```

All we have done so far is set up the apparatus for our simulation. Let's create some power generators - 5 wind turbines and 5 solar farms. Here we store these in lists.

```
turbines = []
solar_farms = []
for i in range(0,5):
    turbines.append(power_generator())
```

```
for i in range(0,5):
    solar_farms.append(power_generator())
```

Let's also set up some customers - a heavy-power user factory, and ten farms which use a smaller amount of power.

```
factory = power_user(300)
farms = []
for i in range(10):
    farms.append(power_user(100))
```

Let's run a daily cycle for a year. Each day we start the `grid` level at `0`, generate the day's power and then invite customers to use it. We use the return values from each customer to decide whether to carry on trying to draw power from the grid. (This doesn't quite work, because there might not be enough to power the factory but still enough to power some of the farms, but it's close enough for an approximate model here.)

```
for i in range(365):
    grid = 0

    # generate power
    for turbine in turbines:
        turbine.generate(i)
    for solar_farm in solar_farms:
        solar_farm.generate(i)
    print(f"Start day {i}. Grid power level: {grid}.")

    # use power
    if factory.consume():
        for farm in farms:
            if not farm.consume():
                break
    print(f"End day {i}. Grid power level: {grid}.")
```

If you run this, you might find that some days there is enough power and some days there are 'brown outs'. We can count how many brown out days there are by starting a variable before our loop

```
brown_out_days = 0
```

We would increment this variable when generating power if there isn't enough to power one of our users. For example, we could do it like this:

```
    # use power
    if factory.consume():
        for farm in farms:
            if not farm.consume():
                brown_out_days += 1
                break
    else:
        brown_out_days += 1
```

Lastly, we report the number of `brown_out_days` at the end of the program.

```
print(brown_out_days)
```

One problem with renewable energy generation is that this is variable. One way to model this might be to say we generate more energy in the middle of the year than in the winter. In the winter, there is less energy reaching us from the sun and there are more stormy days when wind turbines cannot be safely used. To simulate this, we use a sine wave based on the length of the year. We should still add some randomness, otherwise we will get the same result every day.

Try replacing the random power generation with this line.

```
grid += round(50*(math.sin(2*math.pi/365*(day-81.75))+1)) + randint(0,250)
```

One way to deal with variable power distribution is by storing power in batteries, so excess power from one day can be used in future days.

To do this in our simulation, we can introduce a new class `power_store`. This has a capacity, the maximum amount of energy it can use, and two methods - one to charge from the excess power in the grid, and the other to provide stored power on demand.

```
class power_store:
    def __init__(self,capacity):
        self.capacity = capacity
        self.level = 0

    def charge(self):
        global grid

        if grid > 0: # we charge
            if (self.capacity - self.level) > grid: # spare capacity
                self.level += grid
                grid = 0
            else:
                grid -= self.capacity - self.level
                self.level = self.capacity

    def decharge(self,amount):
        if self.level >= amount:
            self.level -= amount
            return amount
        else:
            power = self.level
            self.level = 0
            return power
```

Let's set up ten power storage facilities on our grid.

```
stores = []
for i in range(10):
    stores.append(power_store(100))
```

At the end of our daily loop, once all customers have been served, we charge the batteries with any remaining power up to their capacity.

```
for battery in stores:
    battery.charge()
```

To keep track of what is happening, let's report at the end of the loop the total energy stored and any excess power that could not be stored.

```
overall_storage = 0
for battery in stores:
    overall_storage += battery.level
print(f"Stored in batteries: {overall_storage}")

if grid > 0:
    print(f"Unstored power: {grid}")
```

If you run this, you'll find over the first few days your batteries charge up, and then they just hold that charge. This is because we haven't told the customers how to use the battery charge yet.

To do this requires a modification to the `consume` method in `power_user` which says if there isn't enough energy in the grid, try drawing from the batteries before reporting a brown out.

```
def consume(self):
    global grid, stores

    if grid >= self.level:
        grid -= self.level
        return True
    else:
        target = self.level - grid
        grid =0
        for battery in stores:
            if target > 0:
                target -= battery.decharge(target)
        if target > 0:
            print("Not enough power")
            return False
        else:
            return True
```

We can plot the daily production and storage of energy in our grid. To do this, before the loop we set up lists to fill with these values.

```
days = []
daily_production = []
daily_storage = []
```

At the start of the loop, we append the current day.

```
days.append(i)
```

Once we have finished generating today's power, we append the current value of `grid` (which, remember, started the day at zero).

```
daily_production.append(grid)
```

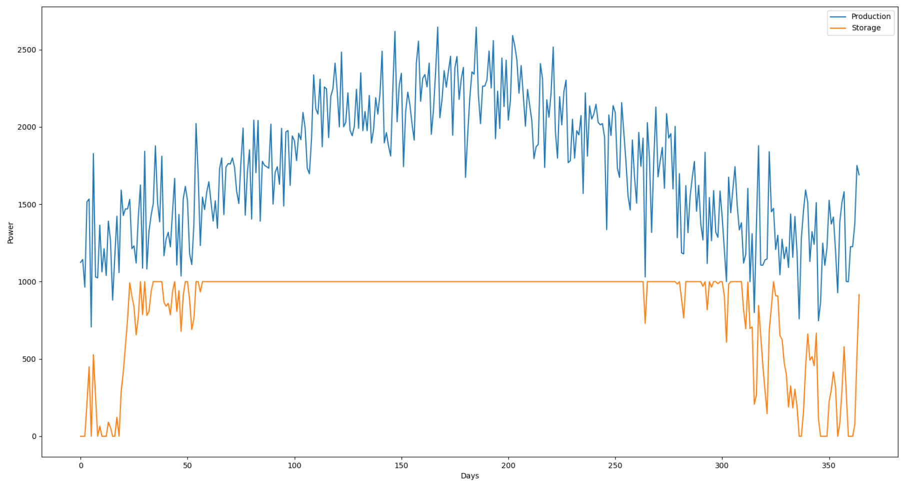
And, once we have finished storing excess power in batteries, we append this value to its list.

```
daily_storage.append(overall_storage)
```

After the loop, we can plot these values.

```
plt.plot(days,daily_production)
plt.plot(days,daily_storage)
plt.xlabel('Days')
plt.ylabel('Power')
plt.legend(['Production', 'Storage'])
plt.show()
```

Running this, I got a plot like this. It simulates well the variable levels of energy production through the year. It also shows the batteries are full and not being used during the summer, but how much they are being used to prop up the grid during the winter. This simulation reported 21 days with brown outs, compared to around 60 days before I introduced battery storage.



You could use this simulation to experiment with the minimum number of battery storage facilities needed to prevent brown outs, for example.