Last week’s exercise

Object-oriented programming

We model “objects” as instances of classes, where we for each class determine:

- Which information is associated with instances? – Instance variables
- What can be done with instances? – Methods

Object-oriented programming

We model “objects” as instances of classes, where we for each class determine:

- Which information is associated with instances? – Instance variables
- What can be done with instances? – Methods

Collectively Attributes. (Sometimes used just for instance variables.)

Improvements to balls.py?

Improvements to `balls.py`?

In my `balls6.py` now the class `Ball` has this `move` method:

```
...
def move(self):
    min_x = min_y = self.radius
    max_x = SCREEN_WIDTH - self.radius
    max_y = SCREEN_HEIGHT - self.radius
    self.x = constrain(min_x,
                       self.x + self.dx,
                       max_x)
    self.y = ...
    if self.x in (min_x, max_x):
        self.dx = -self.dx
    ...
```

Improvements to `balls.py`?

In my `balls6.py` now the class `Ball` has this `move` method:

```
...
def move(self):
    min_x = min_y = self.radius
    max_x = SCREEN_WIDTH - self.radius
    max_y = SCREEN_HEIGHT - self.radius
    self.x = constrain(min_x,
                       self.x + self.dx,
                       max_x)
    self.y = ...
    if self.x in (min_x, max_x):
        self.dx = -self.dx
    ...
```

The min/max values!

The min/max values could be computed once for all (if the balls will keep the same radius).

```
...
def move(self):
    self.x = constrain(self.min_x,
                       self.x + self.dx,
                       self.max_x)
    self.y = ...
    if self.x in (self.min_x, self.max_x):
        self.dx = -self.dx
    ...
```

The min/max values!

Set these in `__init__` (just once) instead.

```
def __init__(self, x, y, radius):
    self.x = x
    self.y = y
    self.radius = radius
    self.min_x = self.min_y = radius
    self.max_x = SCREEN_WIDTH - radius
    self.max_y = SCREEN_HEIGHT - radius
    self.randomize()
```

balls6_special.py

`balls6_special.py`

- Note how they're added to `balls` like "normal" balls
- So moved like the others and drawn like the others.
- That works as long there is a method `move` and attributes `x`, `y`, `color` and `radius`.
- These have all the same colour and radius. Unnecessary to set that individually for each instance?

DRY = Don't repeat yourself!

Class variables

```
class SpecialBall:
    color = 250, 20, 20
    radius = 8

    def __init__(self, x, y):
        ...
```

```
>>> b = SpecialBall(100, 200)
>>> b.color
(250, 20, 20)
>>>
```

Class variable (also static variable) shared by all instances of the class. (As opposed to an *instance variable* which is specific to each instance.)

Class variables & methods

(There are also class *methods* and static *methods*. These are actually different, and will not be used in this (part of the) course, where we'll just use the "normal" *instance methods*.)

SpecialBalls are a lot like Balls

SpecialBalls are a lot like Balls

The `move` method should be identical. But copying it is not good. (DRY!)

SpecialBalls are a lot like Balls

The `move` method should be identical. But copying it is not good. (DRY!)

With inheritance one class (the *child*) inherits some of its attributes from another class (the *parent*).

Parent class = Superclass.
Child class = Subclass.

SpecialBalls are a lot like Balls

The `move` method should be identical. But copying it is not good. (DRY!)

With inheritance one class (the *child*) inherits some of its attributes from another class (the *parent*).

Parent class = Superclass.
Child class = Subclass.

The subclass can override any method of it. Then its own (more specialized) version should be used.

Actual full definition of SpecialBall

```
class SpecialBall(Ball):
    color = 250, 20, 20
    radius = 8

    def __init__(self, x, y):
        self.x, self.y = x, y
        speed = randint(4, 6)
        self.dx = choice((-1, 1)) * speed
        self.dy = choice((-1, 1)) * speed
        self.set_minmax()
```

Actual full definition of SpecialBall

```
class SpecialBall(Ball):
    color = 250, 20, 20
    radius = 8

    def __init__(self, x, y):
        self.x, self.y = x, y
        speed = randint(4, 6)
        self.dx = choice((-1, 1)) * speed
        self.dy = choice((-1, 1)) * speed
        self.set_minmax()
```

(`randint` and `choice` are imported from `random`)

Public/non-public attributes

Public/non-public attributes

A programming environment can hide attributes with names *beginning* with underscore (like `_min_x`).

These are called non-public or private attributes.

Used only by the object itself (in its different methods).

Public/non-public attributes

A programming environment can hide attributes with names *beginning* with underscore (like `_min_x`).

These are called non-public or private attributes.

Used only by the object itself (in its different methods).

In some other programming languages private means that you *can't* access it except from inside that class.

Public/non-public: example

Especially important when code is reused, like in a module used by several.

```
import random

print(random.random())
print(random.choice([1, 'two', False]))
```

Public/non-public: example

Especially important when code is reused, like in a module used by several.

```python
import random

print(random.random())
print(random.choice([1, 'two', False]))
```

There is a function `random._sqrt` but it shouldn't be used.

Public/non-public: example

Especially important when code is reused, like in a module used by several.

```
import random

print(random.random())
print(random.choice([1, 'two', False]))
```

There is a function `random._sqrt` but it shouldn't be used.

Evidently a $\sqrt{\ }$ function was useful for those who wrote `random`, but in the next version it might be gone. It's not part of what that module provides.

Public/non-public: example

```python
from textblob import TextBlob

ex = TextBlob("This is an example. Take note.")

ex.tags
ex.sentences
ex.ngrams()
```

Public/non-public: example

```python
from textblob import TextBlob

ex = TextBlob("This is an example. Take note.")

ex.tags
ex.sentences
ex.ngrams()
```

Lots of public attributes. Others are for example `_cmpkey` and `_compare`.

Interface and Information hiding

Interface and Information hiding

> *Our interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object. When we, as the calling object, access these methods, we do not know or care if the television is getting its signal from an antenna, a cable connection, or a satellite dish. We don't care what electronic signals are being sent to adjust the volume, or whether that volume is being output to speakers or a set of headphones. If we open the television to access the internal workings, for example to split the output signal to both external speakers and a set of headphones, we will void the warranty.* *OOP*

Also called Encapsulation.

Checking the type of an object

Checking the type of an object

Not a good idea:

```
def double_if_string(x):
    if type(x) == str:
        return x + x
    else:
        return x
```

Checking the type of an object

Not a good idea:

```
def double_if_string(x):
    if type(x) == str:
        return x + x
    else:
        return x
```

The right way!

```
def double_if_string(x):
    if isinstance(x, str):
        return x + x
    else:
        return x
```

So will work even if some special kind of string has been used.
(Example with textblob!)

Every SpecialBall instance is also a Ball instance

```
>>> b = SpecialBall(100, 100)
>>> isinstance(b, SpecialBall)
True
>>> isinstance(b, Ball)
True
>>>
```

There is a hierarchy of types (also for the standard types).

Example: first_double

Here the intention is to return the first element that is a sequence of length 2.

```
def first_double(L):
    for element in L:
        if len(element) == 2:
            return element
    return None
```

Example: first_double

Here the intention is to return the first element that is a sequence of length 2.

```
def first_double(L):
    for element in L:
        if len(element) == 2:
            return element
    return None
```

Allow elements to be something else (that should be skipped).

Example: first_double

Here the intention is to return the first element that is a sequence of length 2.

```
def first_double(L):
    for element in L:
        if len(element) == 2:
            return element
    return None
```

Allow elements to be something else (that should be skipped).

```
def first_double(L):
    for element in L:
        if (type(element) in (str, list)
              and len(element) == 2):
            return element
    return None
```

How to improve?

Example: first_double

```python
def first_double(L):
    for e in L:
        if isinstance(e, str) or isinstance(e, list):
            if len(e) == 2:
                return e
    return None
```

Example: first_double

```python
def first_double(L):
    for e in L:
        if isinstance(e, str) or isinstance(e, list):
            if len(e) == 2:
                return e
    return None
```

How to improve?

Example: first_double

```python
def first_double(L):
    for e in L:
        if isinstance(e, str) or isinstance(e, list):
            if len(e) == 2:
                return e
    return None
```

How to improve?

What we are really interested in is if it's some kind of sequence so it's possible to check its length.
Actually the `len` function depends on its argument having a method `__len__`.

Example: first_double

So yet another version:

```
def first_double(L):
    for e in L:
        if hasattr(e, '__len__') and len(e) == 2:
            return e
    return None
```

Example: first_double

So yet another version:

```
def first_double(L):
    for e in L:
        if hasattr(e, '__len__') and len(e) == 2:
            return e
    return None
```

By the way, did you notice that earlier I forgot

Example: first_double

So yet another version:

```
def first_double(L):
    for e in L:
        if hasattr(e, '__len__') and len(e) == 2:
            return e
    return None
```

By the way, did you notice that earlier I forgot that there are other sequences than *lists* and *strings*, at least *tuples*?

Example: first_double – or just go ahead!

Example: first_double – or just go ahead!

EAFP = "Easier to Ask for Forgiveness than Permission"

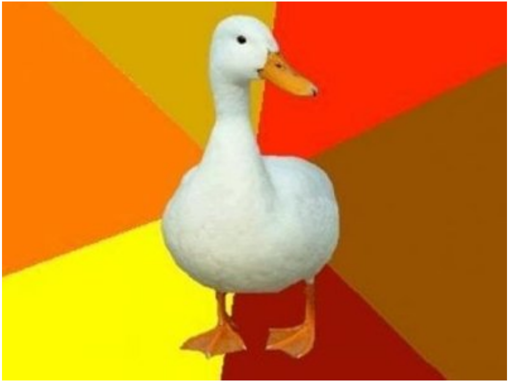
Example: first_double – or just go ahead!

EAFP = "Easier to Ask for Forgiveness than Permission"

```
def first_double(L):
    for element in L:
        try:
            if len(element) == 2:
                return element
        except TypeError:
            pass
    return None
```

Duck typing

Duck typing



If it walks like a duck and it quacks like a duck, then it must be a duck.

In Python it is often not so important which type an object *really* has. The important is what attributes the object has. Because of the saying above this kind of thinking is called duck typing.

Polymorphism

```
for ball in balls:
    ball.move()

player.move()
```

Polymorphism

```
for ball in balls:
    ball.move()

player.move()
```

```
for obj in objects:
    obj.move()
```

There could any kind of objects in there, as long as they all have a method `move`

Polymorphism

```
for ball in balls:
    ball.move()

player.move()
```

```
for obj in objects:
    obj.move()
```

There could any kind of objects in there, as long as they all have a method `move`

Some parts of the program can handle these different objects the same.

Polymorphism

> *In programming languages and type theory, polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.* *[Wikipedia]*

(Like this single method `move` that different objects can treat differently.)

balls6_sleeping.py

balls6_sleeping.py

- Sometimes sleeps for exactly 100 steps in the loop, so it has a counter `self.sleeping` which says how much more it should sleep.
- When waking up it sets a new random direction. That is made into its own method `set_direction`.

balls6_sleeping.py

- Sometimes sleeps for exactly 100 steps in the loop, so it has a counter `self.sleeping` which says how much more it should sleep.
- When waking up it sets a new random direction. That is made into its own method `set_direction`.

So moving is:

```
def move(self):
    if self.sleeping > 0:
        self.sleeping -= 1
        if self.sleeping == 0:
            self.set_direction()
    else:
        ...  # Move as usual
        if random() < 0.01:
            self.sleeping = 100
```

Super

```
class SpecialBall(Ball):
    ...
    def move(self):
        if self.sleeping > 0:
            ...
        else:
            ...   # Use the move method of Ball
            if random() < 0.01:
                self.sleeping = 100
```

Super

```
class SpecialBall(Ball):
    ...
    def move(self):
        if self.sleeping > 0:
            ...
        else:
            super().move()
            if random() < 0.01:
                self.sleeping = 100
```

Super

```python
class SpecialBall(Ball):
    ...
    def move(self):
        if self.sleeping > 0:
            ...
        else:
            super().move()
            if random() < 0.01:
                self.sleeping = 100
```

With `super()` we get access to the super class. (Here `Ball`.)

Multiple Inheritance

It is possible for a class to inherit several other classes, Multiple Inheritance. (We won't do that now.)

Multiple Inheritance

It is possible for a class to inherit several other classes, Multiple Inheritance. (We won't do that now.)

> *As a rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you're probably right.*
>
> *OOP*

Getters and setters . . .

In some programming languages the difference between *public* and *private* is stressed, and attributes are normally kept private, even if they are supposed to be accessed and changed.

```
class Ball:
    def __init__(self, x, y):
        self._x = x
        self._y = y
```

Getters and setters …

In some programming languages the difference between *public* and *private* is stressed, and attributes are normally kept private, even if they are supposed to be accessed and changed.

```
class Ball:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def get_y(self):
        return self._y
```

Getters and setters …

In some programming languages the difference between *public* and *private* is stressed, and attributes are normally kept private, even if they are supposed to be accessed and changed.

```python
class Ball:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def get_y(self):
        return self._y

    def set_x(self, x):
        self._x = x

    def set_y(self):
        self._y = y
```

Getters and setters . . .

In some programming languages the difference between *public* and *private* is stressed, and attributes are normally kept private, even if they are supposed to be accessed and changed.

Using *getters* (*accessors*) and *setters* (*mutators*) like this instead of just

```
class Ball:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

is not the natural way to do this in Python – can be the sign of code translated from for example Java or C++.

```
class Ball:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def get_y(self):
        return self._y

    def set_x(self, x):
        self._x = x

    def set_y(self):
        self._y = y
```

Can lead to longer and harder-to-read code

```
ball.x += ball.dx
```

vs.

```
ball.set_x(ball.get_x() + ball.get_dx())
```

Why do they do that?

There are good reasons.

- Easier to change the program in a way so that old stuff will still work!
- We can add something extra being done.

Why do they do that?

There are good reasons.

- Easier to change the program in a way so that old stuff will still work!
- We can add something extra being done.

Suppose we later decide it's better to store the diameters of the balls instead of their radiuses.
There is no longer any attribute `radius` but we can change definitions to

```
def get_radius(self):
    return self._diameter/2

def set_radius(self, radius):
    self._diameter = 2*radius
```

and old code will still work.

Why do they do that?

There are good reasons.

- Easier to change the program in a way so that old stuff will still work!
- We can add something extra being done.

For example the *setter* could check if the value looks good and give an error message otherwise. Also the *setter* for `_radius` could at the same time change `_min_x` etc.

The Python solution

> *For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.* *PEP 8*

The Python solution is *properties*

```
class Ball:
    ...
    @property
    def radius(self):
        return self._diameter/2
```

The Python solution is *properties*

```
class Ball:
    ...
    @property
    def radius(self):
        return self._diameter/2

    @radius.setter
    def radius(self, radius):
        self._diameter = 2*radius
```

The Python solution is *properties*

```
class Ball:
    ...
    @property
    def radius(self):
        return self._diameter/2

    @radius.setter
    def radius(self, radius):
        self._diameter = 2*radius
```

You don’t need to memorize that now, but you should know that it exists and what it is for.

Lines beginning with `@` are *decorations* of Python functions or classes. It’s a way to change what they did do in various way, and is a bit complicated. (I haven’t shown any examples of this earlier.)

Summary

- Classes can have public and non-public/private attributes. (In Python their names start with underscore, and they are not really private.)
- Only the public contents of a module is part of its official interface.
- Classes can have class variables.
- Classes can inherit other classes. The subclass can define additional attributes and override any existing methods.
- `super()`
- `isinstance`, `hasattr`. Most times don't check if an objects type is equal to something.
- Easier to Ask for Forgiveness than Permission
- Getters and setters, why normally not used in Python for simple public data attributes.