

# Intro to Coding with Python- Classes Pt 1

Dr. Ab Mosca (they/them)

# Plan for Today

- Recap: **lists** and **dictionaries**
- Activity: **functions** vs. **methods**
- Creating our first **class**
  - **attributes**
  - **methods**
  - **self**
  - getting started

# RECAP: using lists and dictionaries

```
def addSong(library):
    # Initialize an empty dictionary
    song = {}

    # Fill in details
    song["title"] = input("Song title: ")
    song["artist"] = input("Artist: ")
    song["album"] = input("Album: ")

    # Append song to library
    library.append(song)
```

# RECAP: lists and dictionaries

```
def printSongs(library):
    # A counter is one way to number the songs
    counter = 0

    # Loop over all the songs in the library
    for song in library:
        counter += 1
        # String formatting to the rescue!
        print("{}: {} by {} ({})".format(counter, song["title"], song["artist"], song["album"]))

    this feels
    a little funny...
```

## Discussion

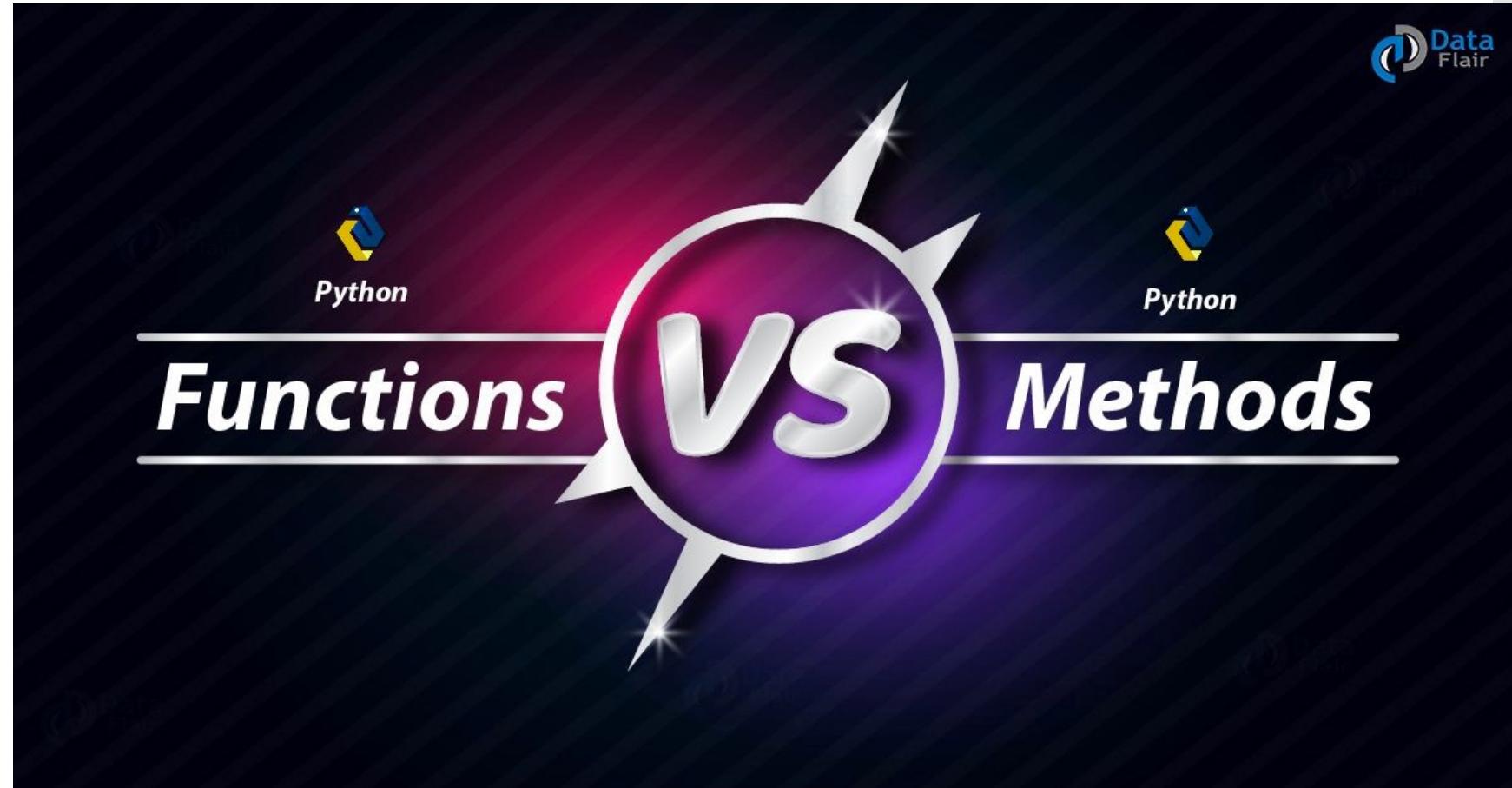
Compare this with other operations we can perform on **lists** and **dictionaries**; what do you **notice**?

<code>animals.append('guinea pig')</code>	<code>pets.count('dog')</code>
<code>vowels.insert(3, 'o')</code>	<code>numbers.reverse()</code>
<code>animals.remove('rabbit')</code>	<code>names.sort()</code>
	<code>numbers.copy()</code>

## Discussion

So what's the difference between  
a **function** and a **method**?

# Activity: functions vs. methods



# Back to the playlist: what we want

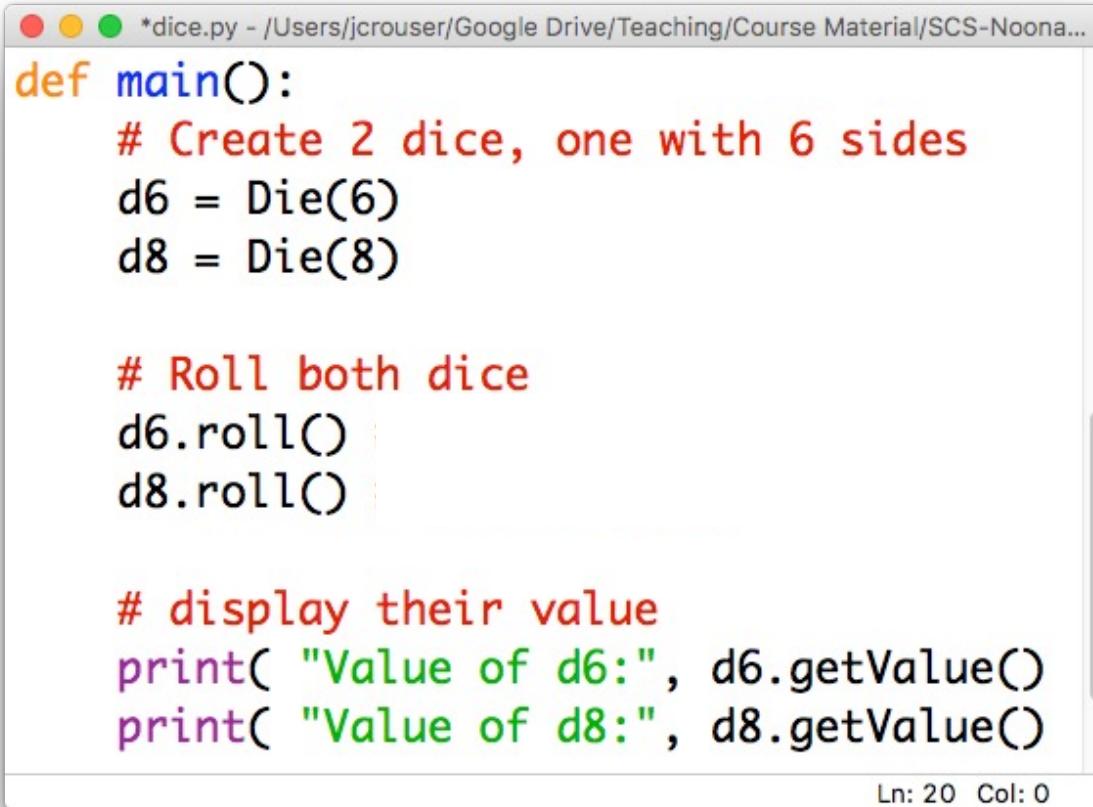
- We'd like to be able to ask a **Song** to **print()** or **play()** itself (since it already has access to all the information)
- That way we don't have to waste time **passing everything** from function to function
- To do this, we'll need a way to combine functions (**methods**) and variables (**attributes**)
- Solution: **classes**



# Building a Die class



How this  
might look



The image shows a screenshot of a Python code editor window titled '\*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...'. The code is color-coded to highlight syntax: blue for `def` and `main`, red for comments, and green for strings. The code defines a main function that creates two dice (one 6-sided, one 8-sided), rolls them, and prints their values.

```
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6)
    d8 = Die(8)

    # Roll both dice
    d6.roll()
    d8.roll()

    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())
```

Ln: 20 Col: 0

Just one  
problem...

```
*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6)
    d8 = Die(8)

    # Roll both dice
    d6.roll()
    d8.roll()

    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())

Ln: 20 Col: 0
```

we need to build it a **blueprint**

1. a way to  
build a Die  
given # sides

```
*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6)
    d8 = Die(8)

    # Roll both dice
    d6.roll()
    d8.roll()

    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())

Ln: 20 Col: 0
```

2. to be able to  
**.roll()**  
them

```
*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6)
    d8 = Die(8)

    # Roll both dice
    d6.roll()
    d8.roll()

    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())

Ln: 20 Col: 0
```

3. to be able to  
**.getValue()**

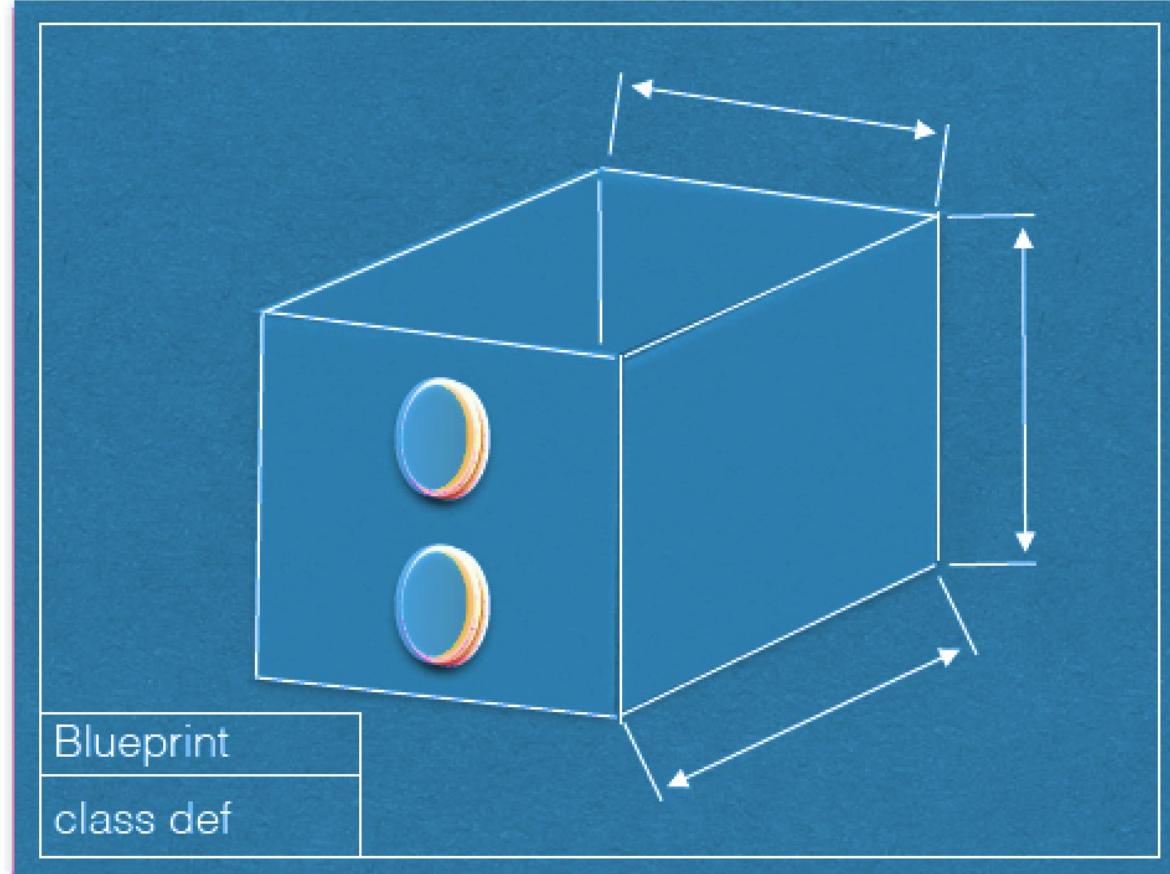
```
*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6)
    d8 = Die(8)

    # Roll both dice
    d6.roll()
    d8.roll()

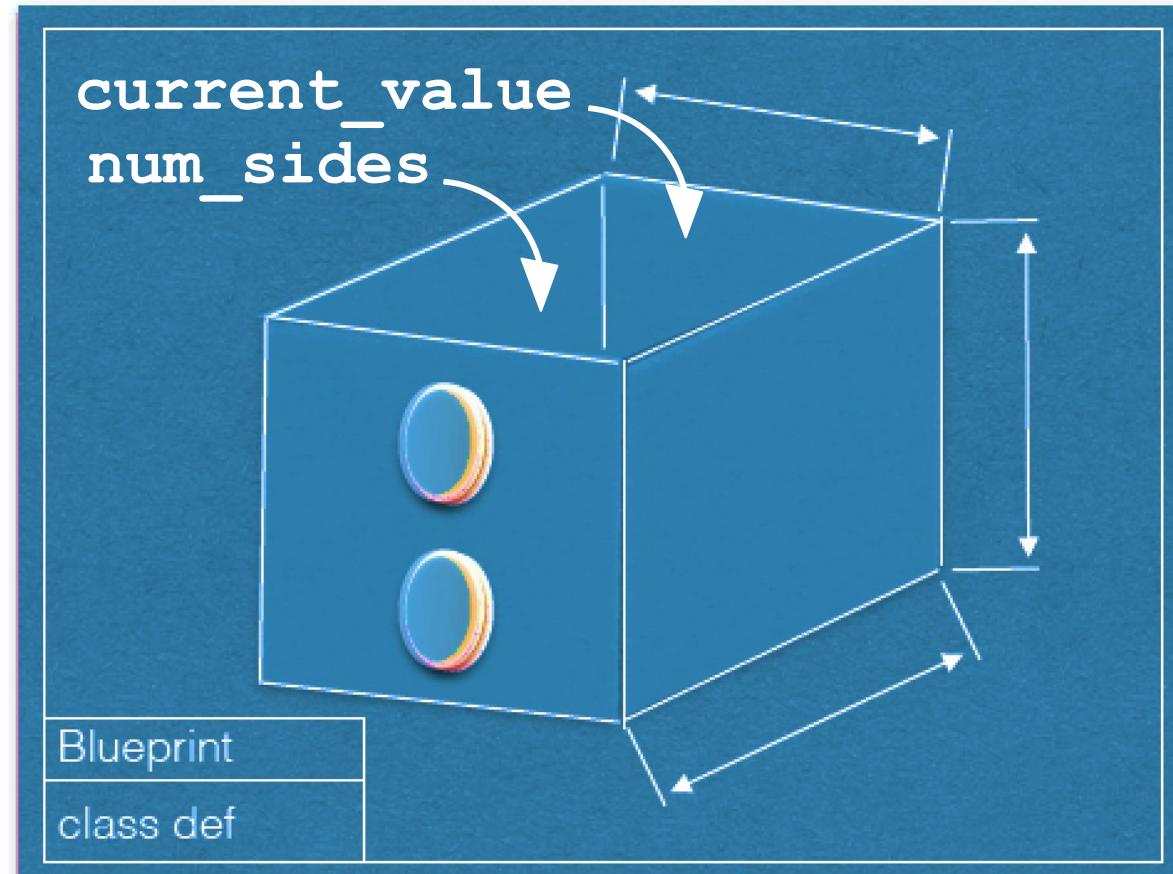
    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())

Ln: 20 Col: 0
```

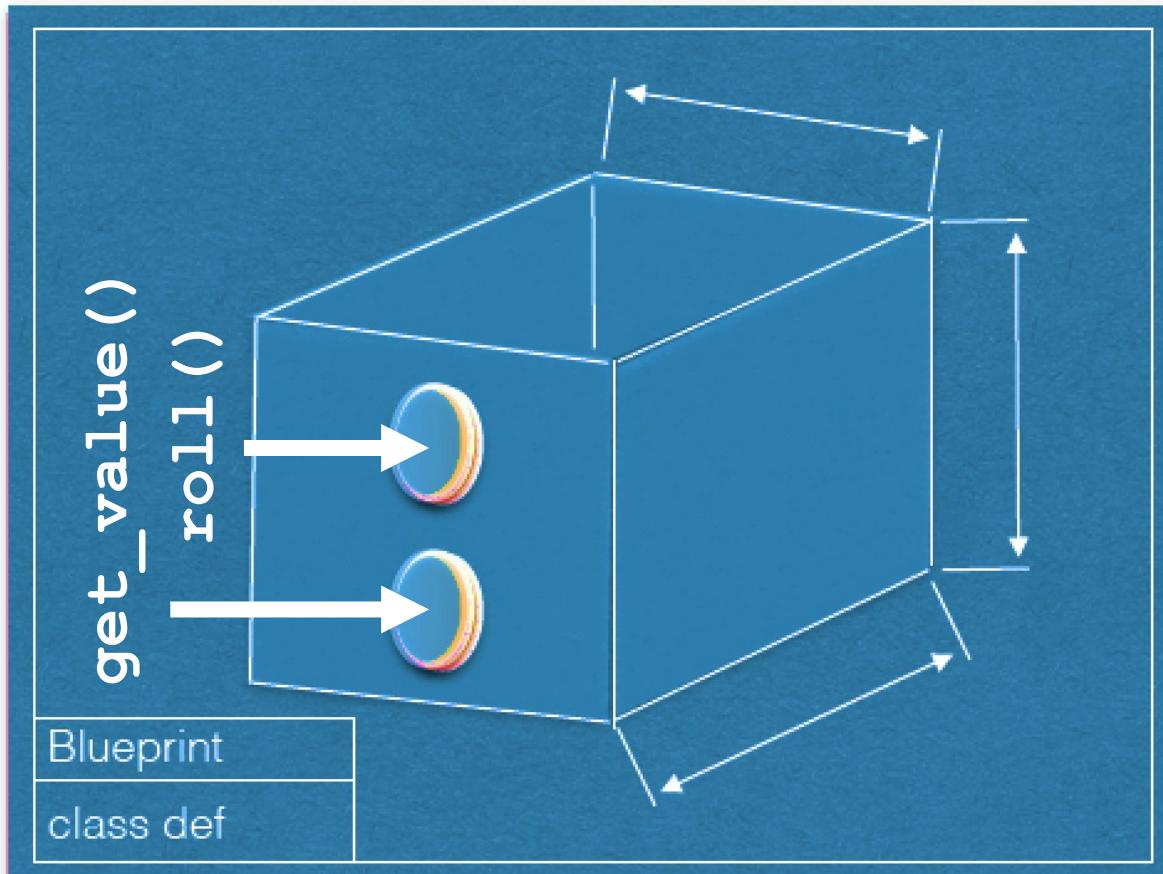
# Blueprint for a Die



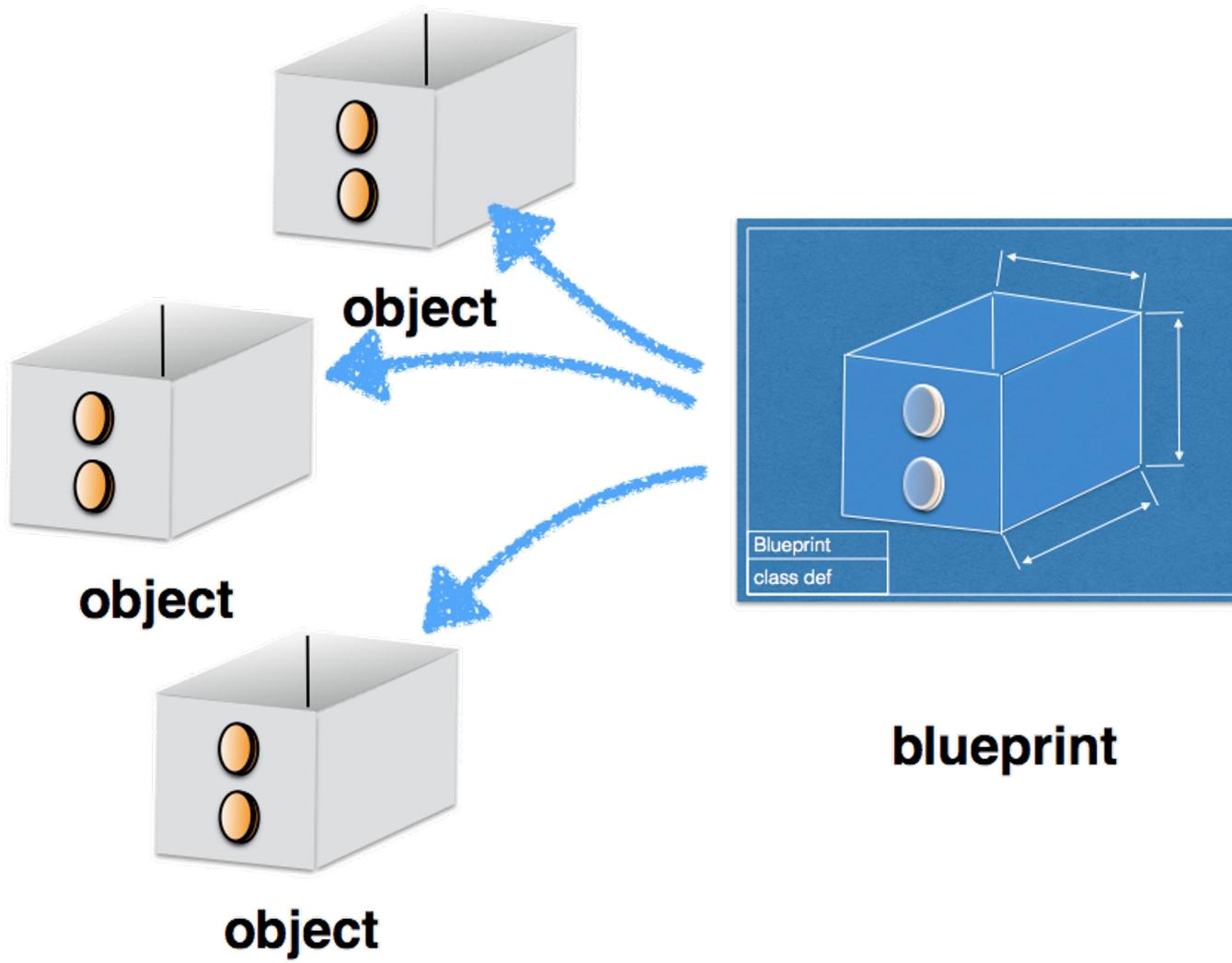
# Blueprint for a Die: attributes



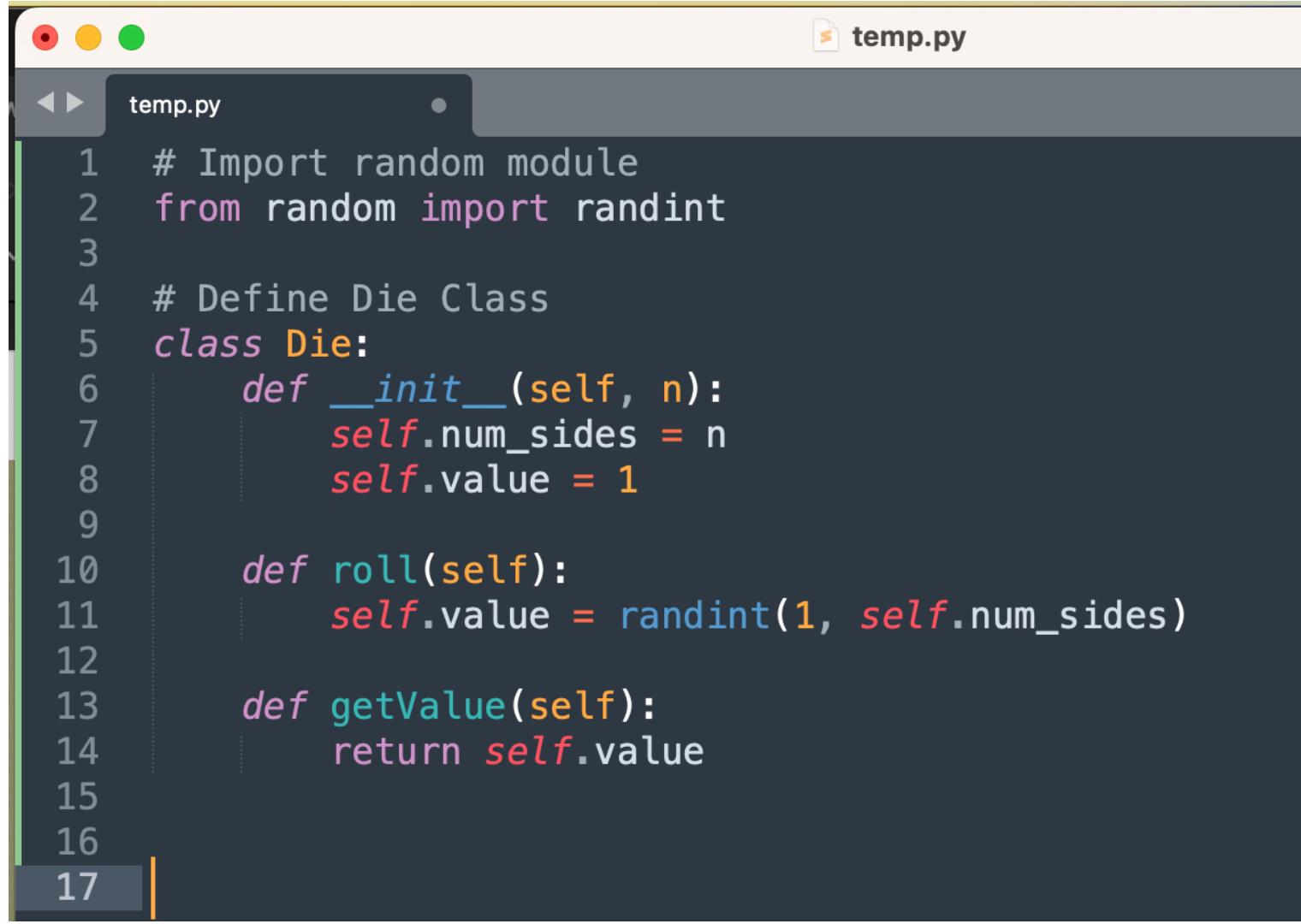
# Blueprint for a Die: methods



Given a  
blueprint,  
replication is  
easy



# Coding the Die class

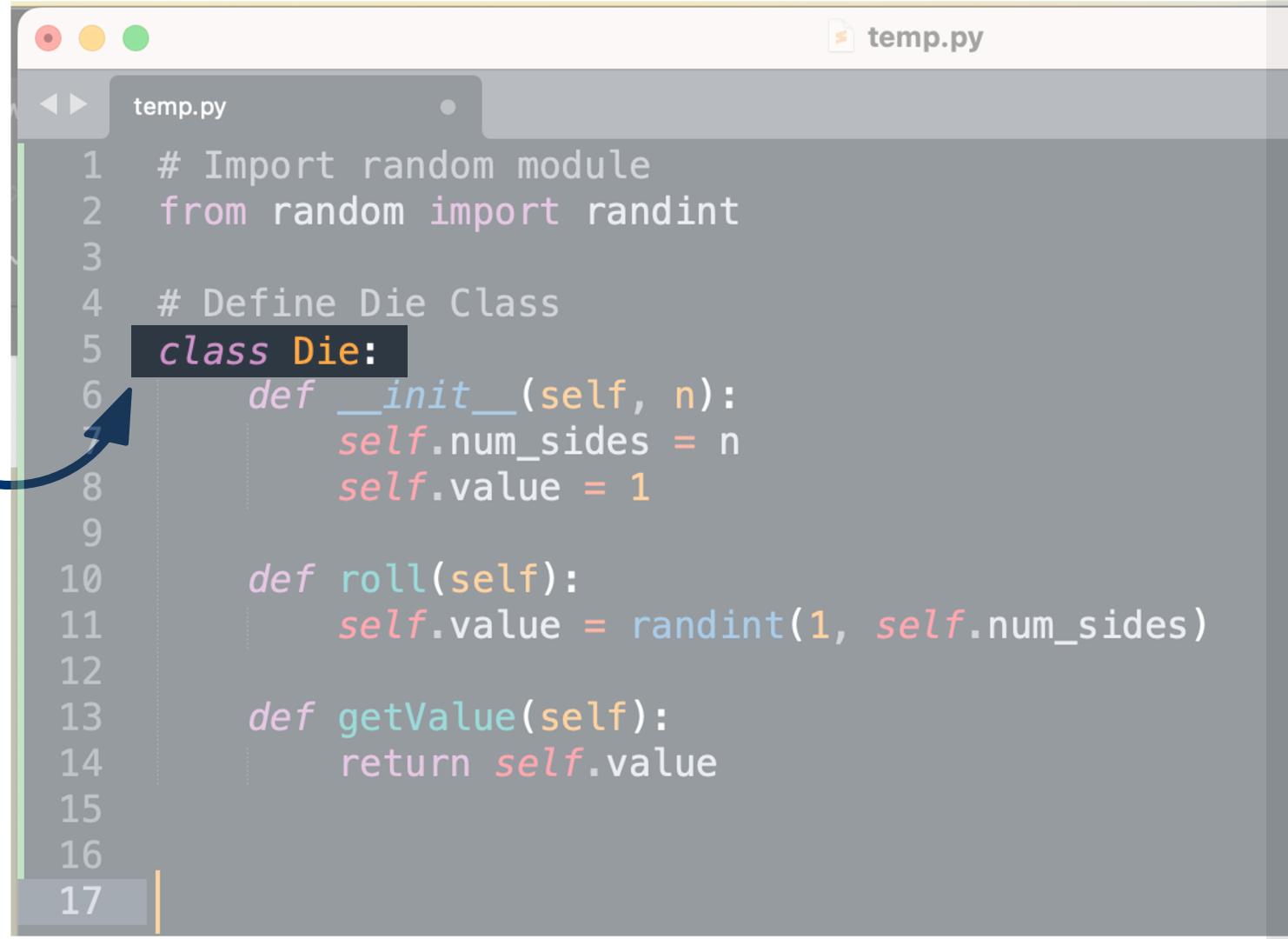


The image shows a screenshot of a Mac OS X desktop environment. A window titled "temp.py" is open, displaying Python code. The code defines a "Die" class with methods for initialization, rolling, and getting the value. The code is color-coded, with keywords like "def" and "class" in blue, and variable names in red.

```
temp.py
1 # Import random module
2 from random import randint
3
4 # Define Die Class
5 class Die:
6     def __init__(self, n):
7         self.num_sides = n
8         self.value = 1
9
10    def roll(self):
11        self.value = randint(1, self.num_sides)
12
13    def getValue(self):
14        return self.value
15
16
17
```

classes are  
defined using  
**class**

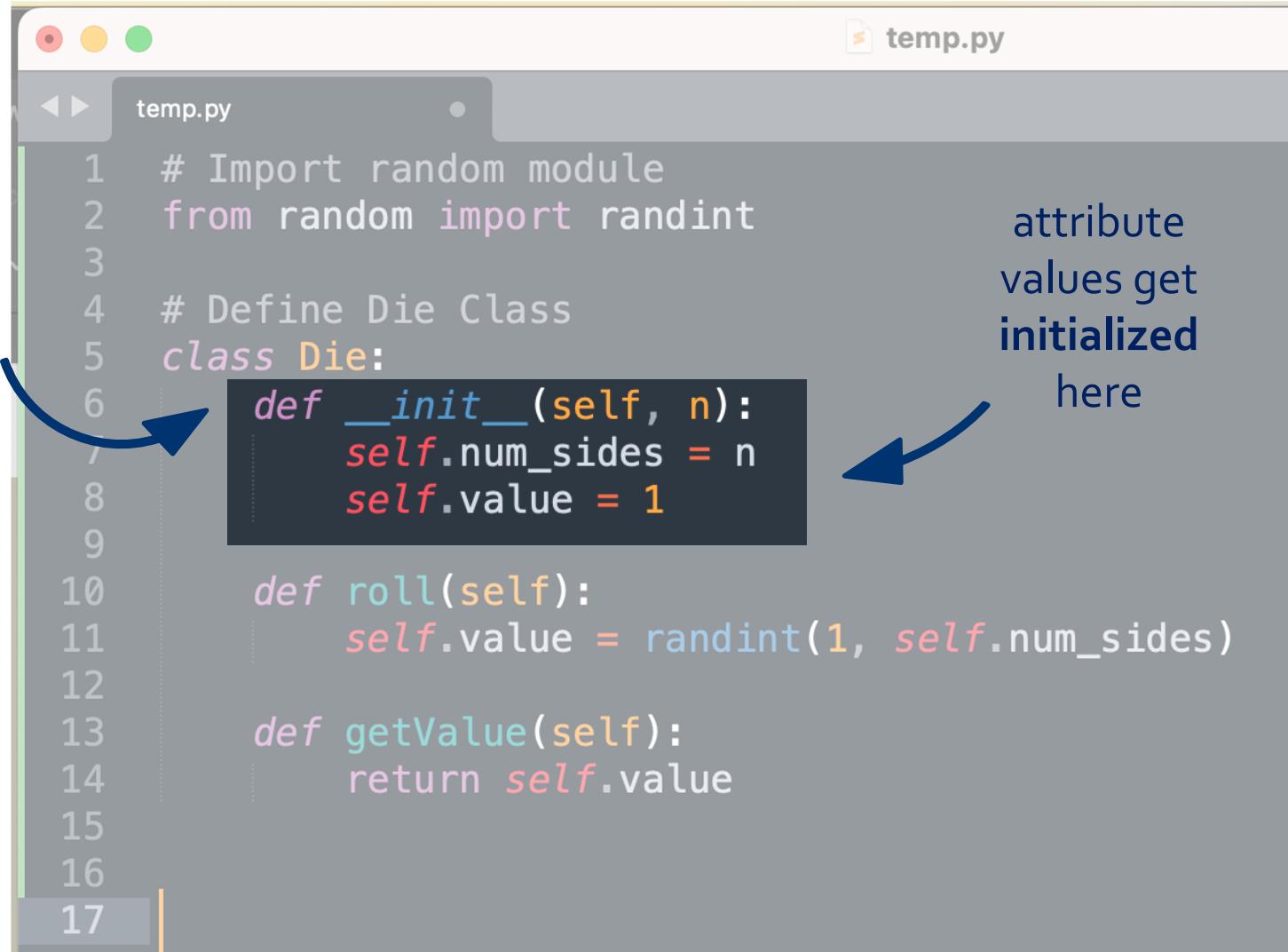
**convention:**  
class names  
start with a  
capital letter



```
temp.py
1 # Import random module
2 from random import randint
3
4 # Define Die Class
5 class Die:
6     def __init__(self, n):
7         self.num_sides = n
8         self.value = 1
9
10    def roll(self):
11        self.value = randint(1, self.num_sides)
12
13    def getValue(self):
14        return self.value
15
16
17
```

# All classes need a constructor

python  
constructors  
are always  
called  
`_init_`

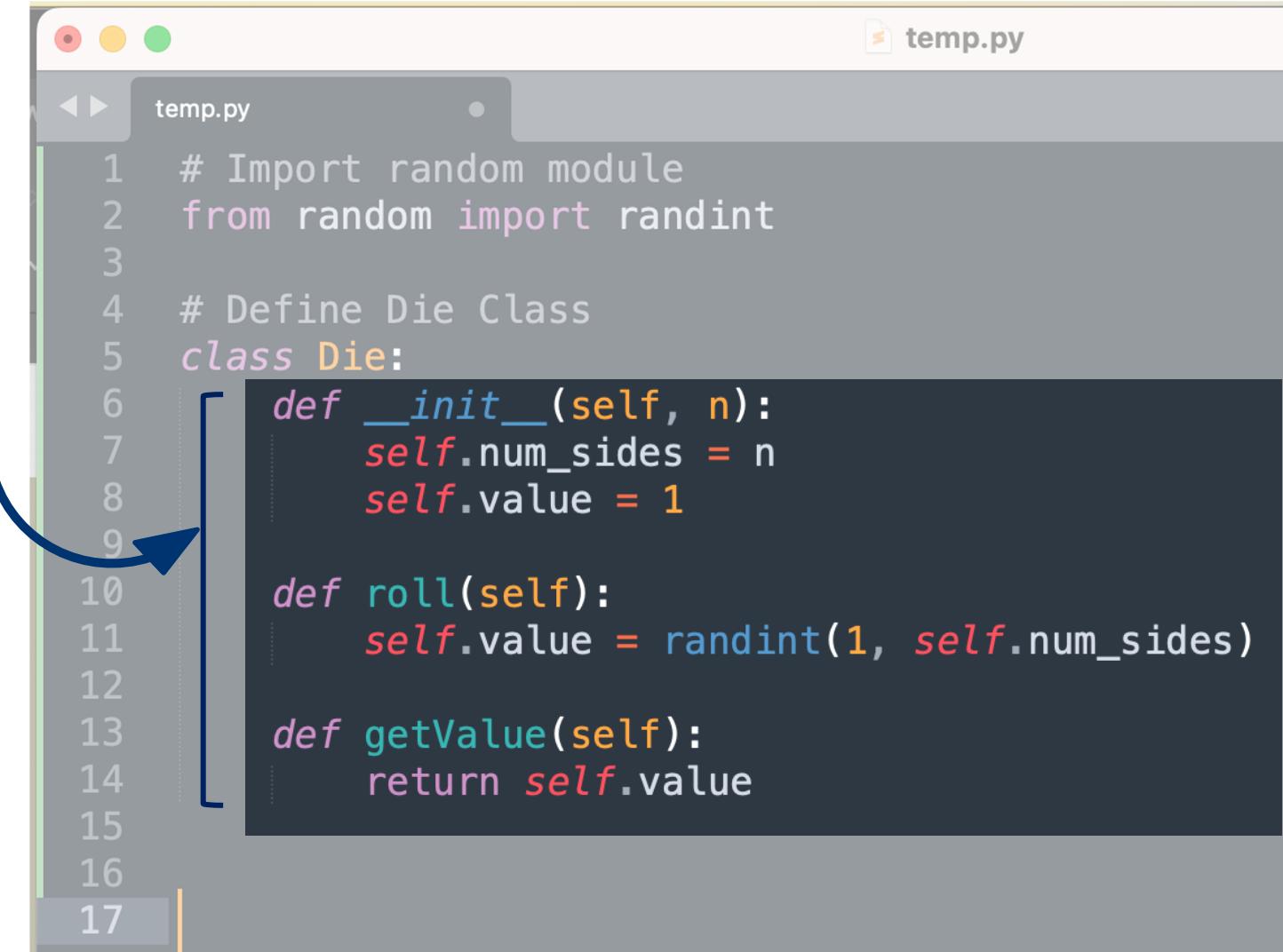


```
temp.py
1 # Import random module
2 from random import randint
3
4 # Define Die Class
5 class Die:
6     def __init__(self, n):
7         self.num_sides = n
8         self.value = 1
9
10    def roll(self):
11        self.value = randint(1, self.num_sides)
12
13    def getValue(self):
14        return self.value
15
16
17
```

attribute values get initialized here

methods are  
**defined** inside  
the **class**

nicely  
indented



```
temp.py
1 # Import random module
2 from random import randint
3
4 # Define Die Class
5 class Die:
6     def __init__(self, n):
7         self.num_sides = n
8         self.value = 1
9
10    def roll(self):
11        self.value = randint(1, self.num_sides)
12
13    def getValue(self):
14        return self.value
15
16
17
```

Question:  
what's with all  
the **selfs**?



The image shows a Mac OS X desktop environment with a single open window titled "temp.py". The window contains Python code for a "Die" class. The code imports the "random" module and defines a "Die" class with three methods: \_\_init\_\_, roll, and getValue. The variable "self" is used throughout the code to refer to the current instance of the class. The code is highlighted with syntax coloring, and the variable "self" is highlighted in yellow across all occurrences.

```
# Import random module
from random import randint

# Define Die Class
class Die:
    def __init__(self, n):
        self.num_sides = n
        self.value = 1

    def roll(self):
        self.value = randint(1, self.num_sides)

    def getValue(self):
        return self.value
```

# Question: what's with all the **self**s?

```
temp.py
```

```
# Import random module
from random import randint

# Define Die Class
class Die:
    def __init__(self, n):
        self.num_sides = n
        self.value = 1

    def roll(self):
        self.value = randint(1, self.num_sides)

    def getValue(self):
        return self.value
```

when attached to a variable, **self** makes the variable a "member" of the **object** (i.e. the **object** owns it)

# Question: what's with all the `self`s?

```
temp.py
```

```
# Import random module
from random import randint

# Define Die Class
class Die:
    def __init__(self, n):
        self.num_sides = n
        self.value = 1

    def roll(self):
        self.value = randint(1, self.num_sides)

    def getValue(self):
        return self.value
```

every method in a class  
**automatically** gets passed  
a reference to the object  
as its first parameter

Again, this  
happens  
automatically

```
*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6) ←
    d8 = Die(8) ← we don't put the self reference
                    into any of the method calls
    # Roll both dice
    d6.roll()
    d8.roll() ←

    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())
Ln: 20 Col: 0
```

But the effect  
is really cool

```
*dice.py - /Users/jcrouser/Google Drive/Teaching/Course Material/SCS-Noona...
def main():
    # Create 2 dice, one with 6 sides
    d6 = Die(6) # Die.__init__(6)
    d8 = Die(8) # Die.__init__(8)

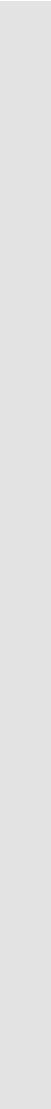
    # Roll both dice
    d6.roll() # Die.roll(d6)
    d8.roll() # Die.roll(d8)

    # display their value
    print("Value of d6:", d6.getValue())
    print("Value of d8:", d8.getValue())

Ln: 20 Col: 0
```

# 15-minute exercise: Classy Playlist

Define Playlist class. Make sure it has methods to ADD, REMOVE, PRINT, or SEARCH



What did you come up with?