RECORDING IN-PROGRESS

Day 9: Classes

Thursday, 7.22.21

RECORDING IN-PROGRESS

Agenda

- Daily check-in
  - Announcements
  - Review solution to Lab 8
- Recap from yesterday
- New topic: Classes
- Live coding

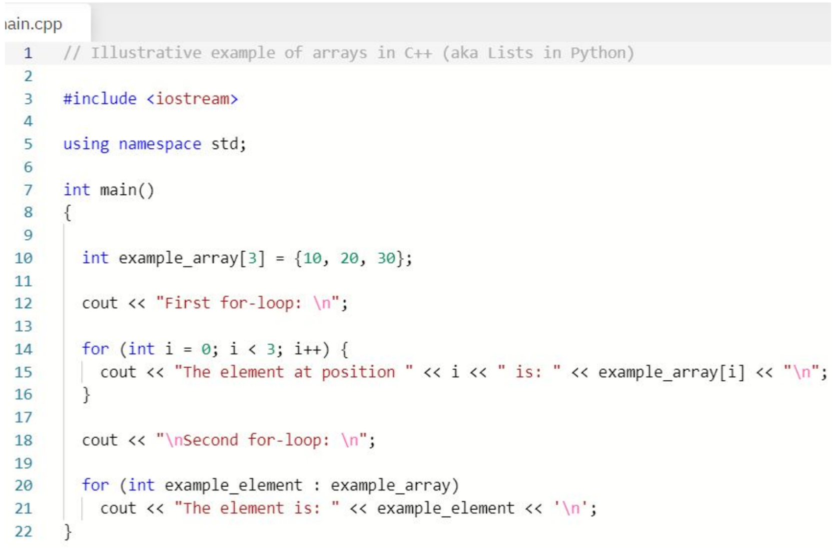
Daily check-in

Check in

- Questions?
- Announcements
  - Our tech trek today will be from **12:30 - 1:30pm EST**, in Zoom as usual, **guest speaker Connor Gramazio will be joining us from Amazon, Cambridge, MA**
  - Connor is a Senior Software Engineer at Amazon working on the *Alexa AI* system ("ask alexa"). His main area of 'research' at Amazon is in NLP, and he uses a lot of C++ and Python for his work at Amazon. Did his undergraduate @Tufts, MS/PhD @Brown.
- Lab 8: Machine Learning with scikit-learn
  - Brawner won't be here to review solution, but Brooks can answer most of your questions
  - Anything we don't know from Brawner's lab you can ask him today during the afternoon lab, or email him:

RECAP

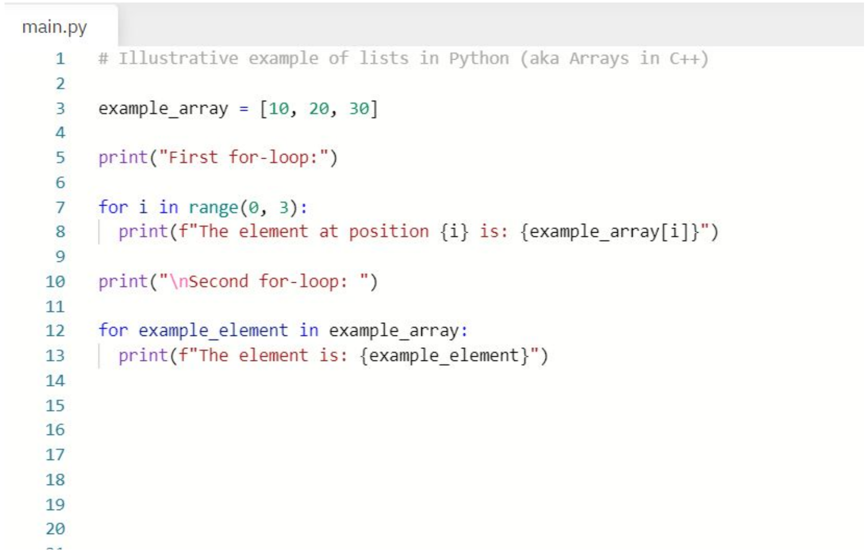
C++ vs. Python



ain.cpp

```cpp
// Illustrative example of arrays in C++ (aka Lists in Python)

#include <iostream>

using namespace std;

int main()
{

  int example_array[3] = {10, 20, 30};

  cout << "First for-loop: \n";

  for (int i = 0; i < 3; i++) {
    cout << "The element at position " << i << " is: " << example_array[i] << "\n";
  }

  cout << "\nSecond for-loop: \n";

  for (int example_element : example_array)
    cout << "The element is: " << example_element << '\n';
}
```

C++ version

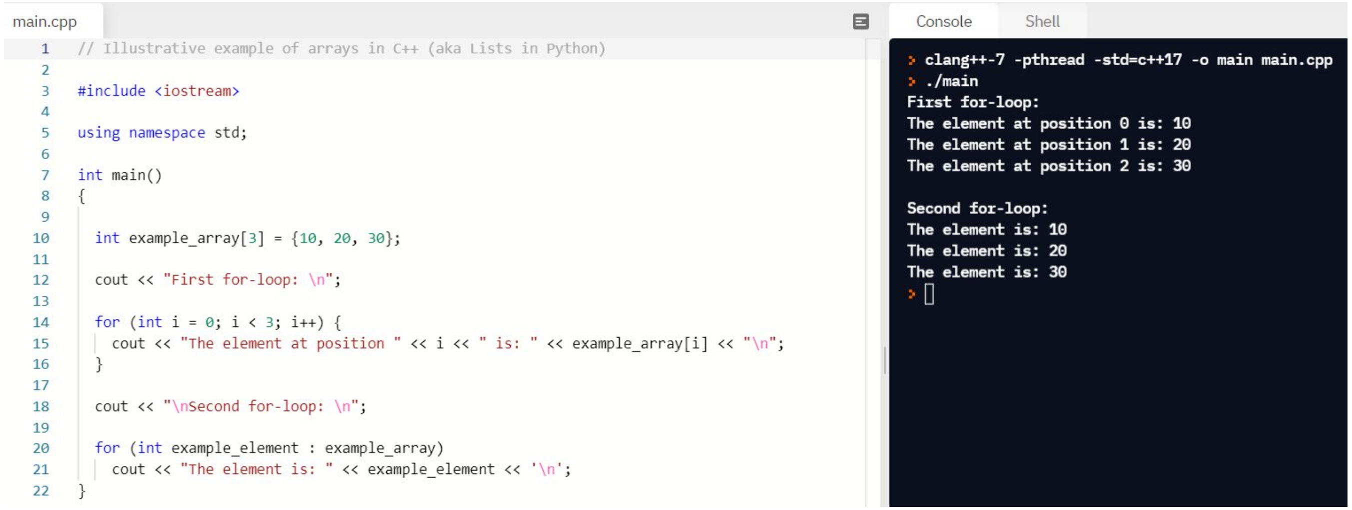


main.py

```python
# Illustrative example of lists in Python (aka Arrays in C++)

example_array = [10, 20, 30]

print("First for-loop:")

for i in range(0, 3):
  print(f"The element at position {i} is: {example_array[i]}")

print("\nSecond for-loop: ")

for example_element in example_array:
  print(f"The element is: {example_element}")
```

Python version

C++ version with Arrays:

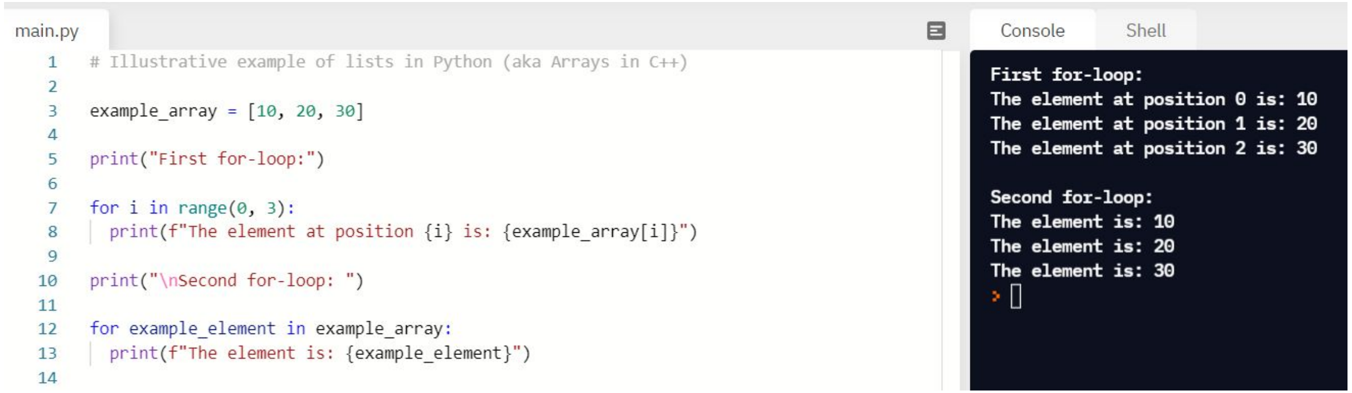

main.cpp

Console Shell

```cpp
// Illustrative example of arrays in C++ (aka Lists in Python)

#include <iostream>

using namespace std;

int main()
{

  int example_array[3] = {10, 20, 30};

  cout << "First for-loop: \n";

  for (int i = 0; i < 3; i++) {
    cout << "The element at position " << i << " is: " << example_array[i] << "\n";
  }

  cout << "\nSecond for-loop: \n";

  for (int example_element : example_array)
    cout << "The element is: " << example_element << '\n';
}
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
First for-loop:
The element at position 0 is: 10
The element at position 1 is: 20
The element at position 2 is: 30

Second for-loop:
The element is: 10
The element is: 20
The element is: 30
> 
```

Python version with Lists:


main.py

Console Shell

```python
# Illustrative example of lists in Python (aka Arrays in C++)

example_array = [10, 20, 30]

print("First for-loop:")

for i in range(0, 3):
  print(f"The element at position {i} is: {example_array[i]}")

print("\nSecond for-loop: ")

for example_element in example_array:
  print(f"The element is: {example_element}")

```

```
First for-loop:
The element at position 0 is: 10
The element at position 1 is: 20
The element at position 2 is: 30

Second for-loop:
The element is: 10
The element is: 20
The element is: 30
> 
```

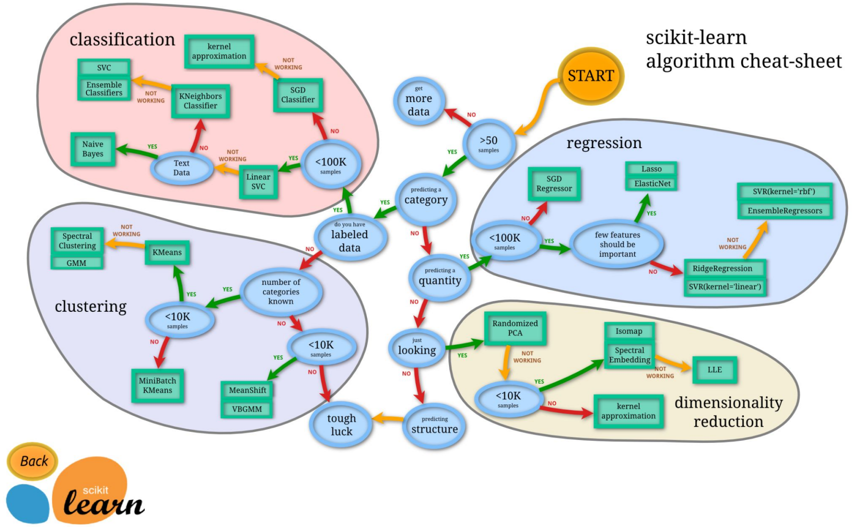
Modules in Python

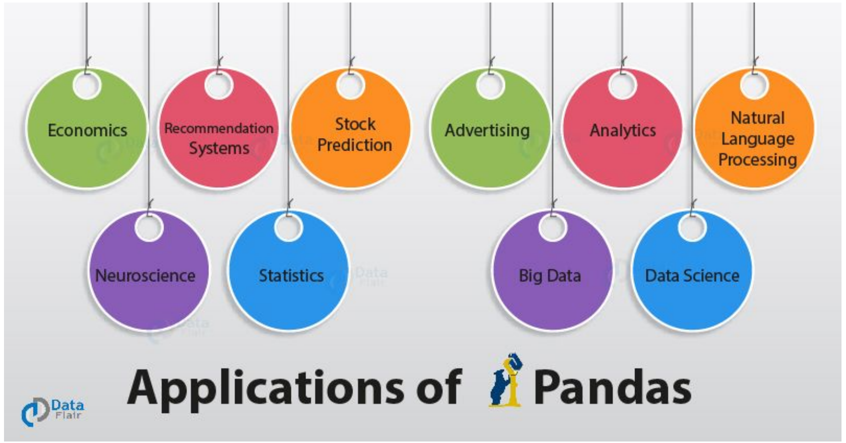
- When we refer to a *module*, we are referring to a single Python file
- Examples of modules we’ve used already:
  - **Time**
    - We could put the program to “sleep” using time.sleep(x), where ‘x’ was the number of seconds we wanted the program to wait
  - **Random**
    - We could randomly pull a number between ‘x’ and ‘y’ using random.randint(x, y)
  - **Math**
    - We could round a decimal value ‘x’ up or down using math.ceil(x) or math.floor(x), respectively

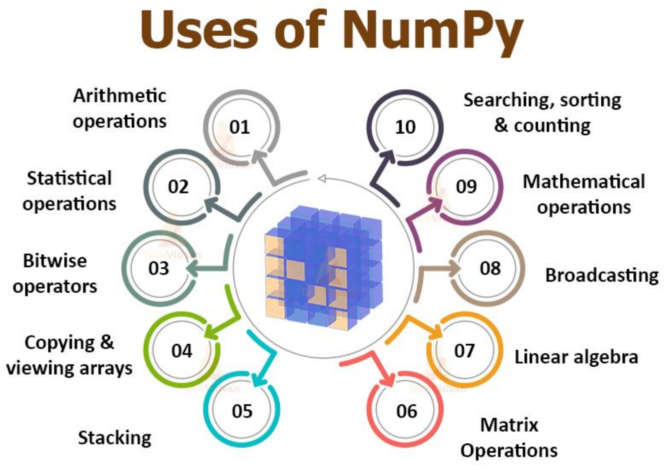
Libraries in Python

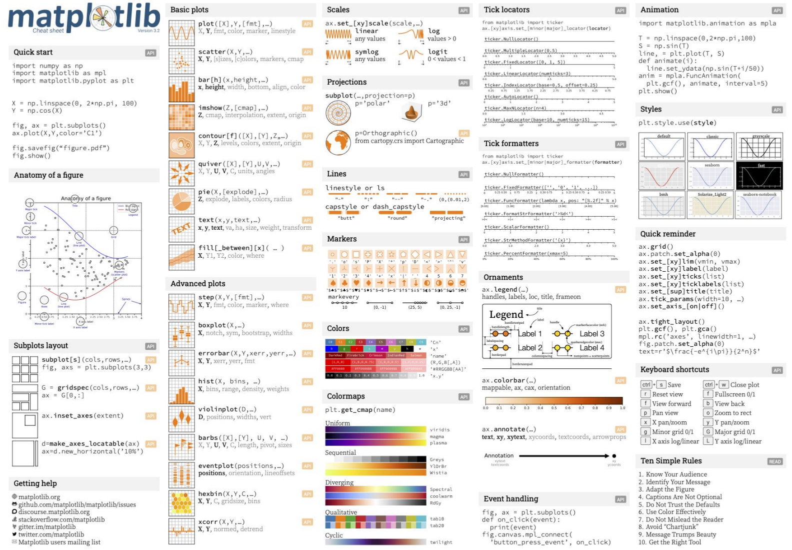
Libraries in Python, on the other hand, are a *collection* of modules.

That is to say, libraries are simply a collection of many Python files (i.e., modules), meaning they likely support a lot more functionality than a single module.








matplotlib Cheat sheet Version 3.2

Quick start

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

X = np.linspace(0, 2*np.pi, 100)
Y = np.cos(X)

fig, ax = plt.subplots()
ax.plot(X,Y,color='C1')

fig.savefig("figure.pdf")
fig.show()
```

Anatomy of a figure

Subplots layout

```
subplot[s](cols,rows,…)
fig, axs = plt.subplots(3,3)

G = gridspec(cols,rows,…)
ax = G[0,:]

ax.inset_axes(extent)

d=make_axes_locatable(ax)
ax=d.new_horizontal('10%')
```

Getting help
- matplotlib.org
- github.com/matplotlib/matplotlib/issues
- discourse.matplotlib.org
- stackoverflow.com/matplotlib
- gitter.im/matplotlib
- twitter.com/matplotlib
- Matplotlib users mailing list

Basic plots

```
plot([X],Y,[fmt],…)
X, Y, fmt, color, marker, linestyle
scatter(X,Y,…)
X, Y, [s]izes, [c]olors, markers, cmap
bar[h](x,height,…)
x, height, width, bottom, align, color
imshow(Z,[cmap],…)
Z, cmap, interpolation, extent, origin
contour[f]([X],[Y],Z,…)
X, Y, Z, levels, colors, extent, origin
quiver([X],[Y],U,V,…)
X, Y, U, V, C, units, angles
pie(X,[explode],…)
Z, explode, labels, colors, radius
text(x,y,text,…)
x, y, text, va, ha, size, weight, transform
fill[_between][x](…)
X, Y1, Y2, color, where
```

Advanced plots

```
step(X,Y,[fmt],…)
X, Y, fmt, color, marker, where
boxplot(X,…)
X, notch, sym, bootstrap, widths
errorbar(X,Y,xerr,yerr,…)
X, Y, xerr, yerr, fmt
hist(X, bins, …)
X, bins, range, density, weights
violinplot(D,…)
D, positions, widths, vert
barbs([X],[Y], U, V, …)
X, Y, U, V, C, length, pivot, sizes
eventplot(positions,…)
positions, orientation, lineoffsets
hexbin(X,Y,C,…)
X, Y, C, gridsize, bins
xcorr(X,Y,…)
X, Y, normed, detrend
```

Scales

```
ax.set_[xy]scale(scale,…)
linear  any values      log    values > 0
symlog  any values      logit  0 < values < 1
```

Projections

```
subplot(…,projection=p)
p='polar'    p='3d'
p=Orthographic()
from cartopy.crs import Cartographic
```

Lines

```
linestyle or ls
capstyle or dash_capstyle
"butt"  "round"  "projecting"
```

Markers

```
markevery
```

Colors

Colormaps

```
plt.get_cmap(name)
```

Uniform: viridis, magma, plasma
Sequential: Greys, YlOrBr, Wistia
Diverging: Spectral, coolwarm, RdGy
Qualitative: tab10, tab20
Cyclic: twilight

Tick locators

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_locator(locator)
ticker.NullLocator()
ticker.MultipleLocator(0.5)
ticker.FixedLocator([0, 1, 5])
ticker.LinearLocator(numticks=3)
ticker.IndexLocator(base=0.5, offset=0.25)
ticker.AutoLocator()
ticker.MaxNLocator(n=4)
ticker.LogLocator(base=10, numticks=15)
```

Tick formatters

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_formatter(formatter)
ticker.NullFormatter()
ticker.FixedFormatter(['', '0', '1', …])
ticker.FuncFormatter(lambda x, pos: "[%.2f]" % x)
ticker.FormatStrFormatter('>%d<')
ticker.ScalarFormatter()
ticker.StrMethodFormatter('{x}')
ticker.PercentFormatter(xmax=5)
```

Ornaments

```
ax.legend(…)
handles, labels, loc, title, frameon
ax.colorbar(…)
mappable, ax, cax, orientation
ax.annotate(…)
text, xy, xytext, xycoords, textcoords, arrowprops
```

Event handling

```
fig, ax = plt.subplots()
def on_click(event):
    print(event)
fig.canvas.mpl_connect(
    'button_press_event', on_click)
```

Animation

```
import matplotlib.animation as mpla

T = np.linspace(0,2*np.pi,100)
S = np.sin(T)
line, = plt.plot(T, S)
def animate(i):
    line.set_ydata(np.sin(T+i/50))
anim = mpla.FuncAnimation(
    plt.gcf(), animate, interval=5)
plt.show()
```

Styles

```
plt.style.use(style)
```

Quick reminder

```
ax.grid()
ax.patch.set_alpha(0)
ax.set_[xy]lim(vmin, vmax)
ax.set_[xy]label(label)
ax.set_[xy]ticks(list)
ax.set_[xy]ticklabels(list)
ax.set_[sup]title(title)
ax.tick_params(width=10, …)
ax.set_axis_[on|off]()

ax.tight_layout()
plt.gcf(), plt.gca()
mpl.rc('axes', linewidth=1, …)
fig.patch.set_alpha(0)
text=r'$\frac{-e^{i\pi}}{2^n}$'
```

Keyboard shortcuts

ctrl + s	Save	ctrl + w	Close plot
r	Reset view	f	Fullscreen 0/1
f	View forward	b	View back
p	Pan view	o	Zoom to rect
x	X pan/zoom	y	Y pan/zoom
g	Minor grid 0/1	G	Major grid 0/1
l	X axis log/linear	L	Y axis log/linear

Ten Simple Rules

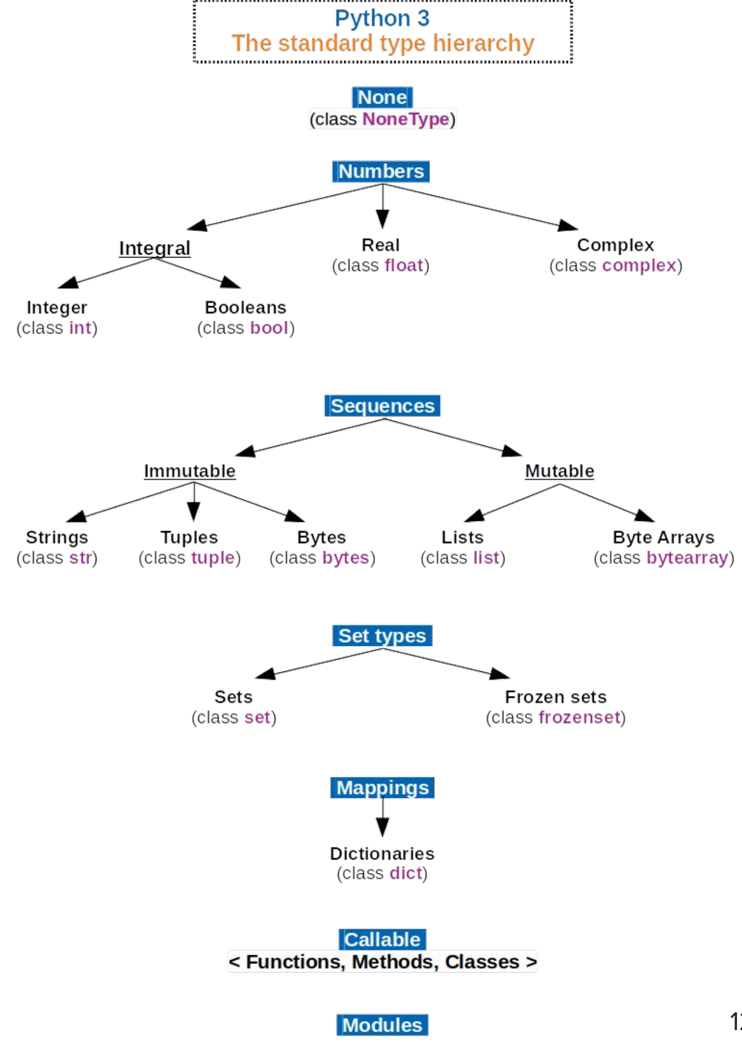
1. Know Your Audience
2. Identify Your Message
3. Adapt the Figure
4. Captions Are Not Optional
5. Do Not Trust the Defaults
6. Use Color Effectively
7. Do Not Mislead the Reader
8. Avoid "Chartjunk"
9. Message Trumps Beauty
10. Get the Right Tool

Tips for working with libraries / packages

- Libraries are fast, helpful, and are generally well documented.
    - Plus, because they're open-source, there's a large community answering questions, debugging problems, posting guides, etc.
- Whenever you want to use a certain function from a library, read the documentation carefully! It's helpful.
    - Read about the input, or parameters, that the function expects.
    - Read about the expected *output* of the function. How does the output depend on the input?
    - Find examples of people using that function on stackoverflow, reddit, whatever
        - Read the questions they post and try to understand the question BEFORE reading the answer

Making our way around Python...

- Numbers ✔
  - Integer
  - Boolean
  - Float
- Sequences ✔
  - Strings
  - Tuples
  - Lists
  - Mutability vs. Immutability
- Sets ✔
- Dictionaries ✔
- Functions ✔
- Modules ✔
- ....last but not least, ***classes***.



New topic: Classes

Classes! But first, Objects.

Before we start talking about *classes*, let’s talk about *objects*.

Under the hood of Python, everything we’ve been working with so far has actually been an *object*. Variables, functions, lists, tuples, dictionaries, sets, etc.

They’re all what’s known as an object.

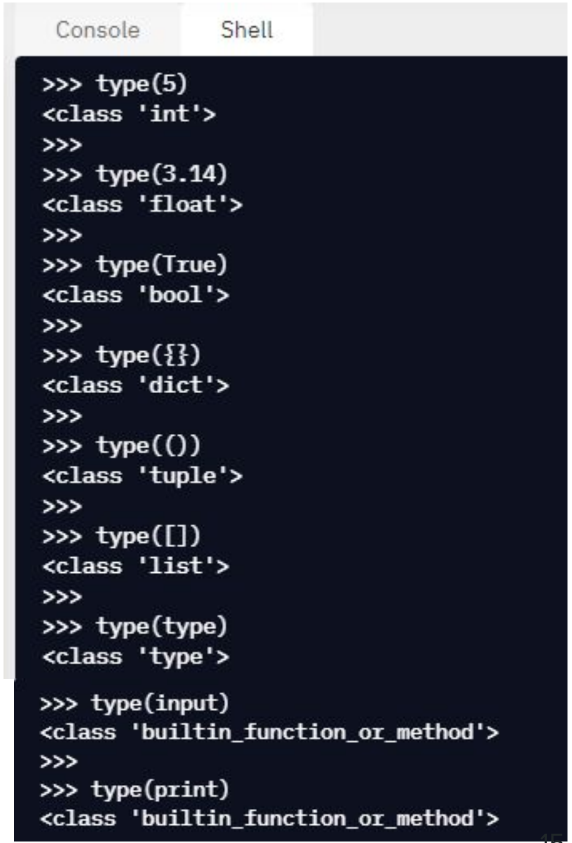
What’s an object? *An object is the collection of various data and functions that operate on those data*. Does anyone have a better definition?

What do objects have to do with classes?

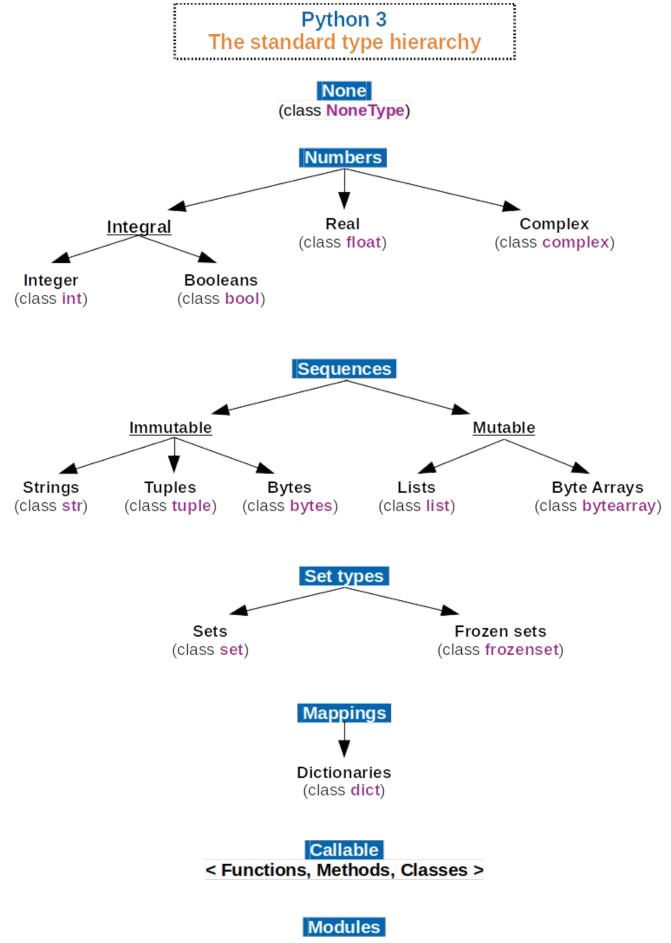
Every object belongs to a *class*.

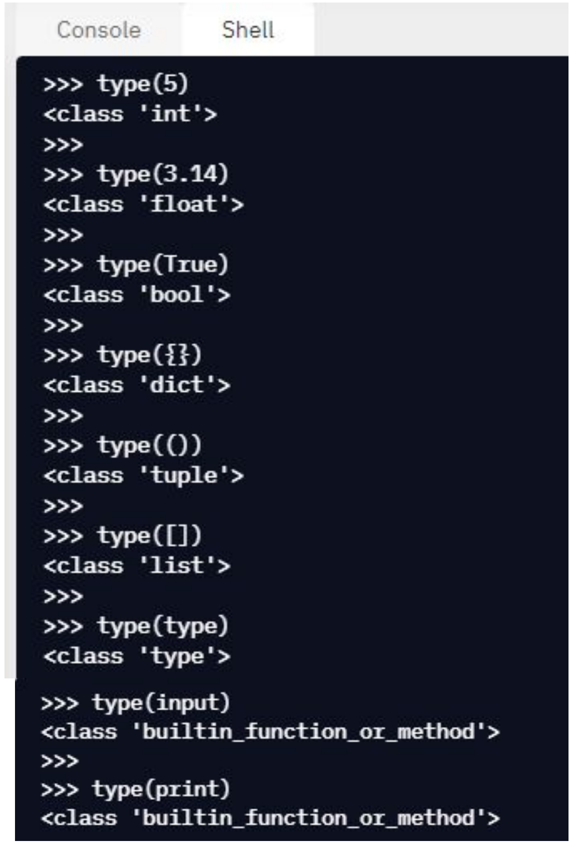
Think back on all the times we've typed "type(X)" into our Python shell. What was the output?

<*class* 'objectClass'>



```
>>> type(5)
<class 'int'>
>>>
>>> type(3.14)
<class 'float'>
>>>
>>> type(True)
<class 'bool'>
>>>
>>> type({})
<class 'dict'>
>>>
>>> type(())
<class 'tuple'>
>>>
>>> type([])
<class 'list'>
>>>
>>> type(type)
<class 'type'>
>>> type(input)
<class 'builtin_function_or_method'>
>>>
>>> type(print)
<class 'builtin_function_or_method'>
```





```
>>> type(5)
<class 'int'>
>>>
>>> type(3.14)
<class 'float'>
>>>
>>> type(True)
<class 'bool'>
>>>
>>> type({})
<class 'dict'>
>>>
>>> type(())
<class 'tuple'>
>>>
>>> type([])
<class 'list'>
>>>
>>> type(type)
<class 'type'>

>>> type(input)
<class 'builtin_function_or_method'>
>>>
>>> type(print)
<class 'builtin_function_or_method'>
```

Just like the chart says! We’ve worked with many built-in classes in Python.

Classes

Classes are sorta similar to functions, such that we can write our own custom version to do whatever we want. However, classes are much more powerful than functions: any class we write can also have its own functions! Think of a class as a *blueprint* for something specific you want to work with.

When we write a class, we can create “instances” of that class by creating / instantiating a new object that *belongs* to that class. Just like when we do:

`x = 5` *in Python* or **`int x = 5;`** *in C++*

We are creating an *instance** of the *integer class* by creating an int object x.

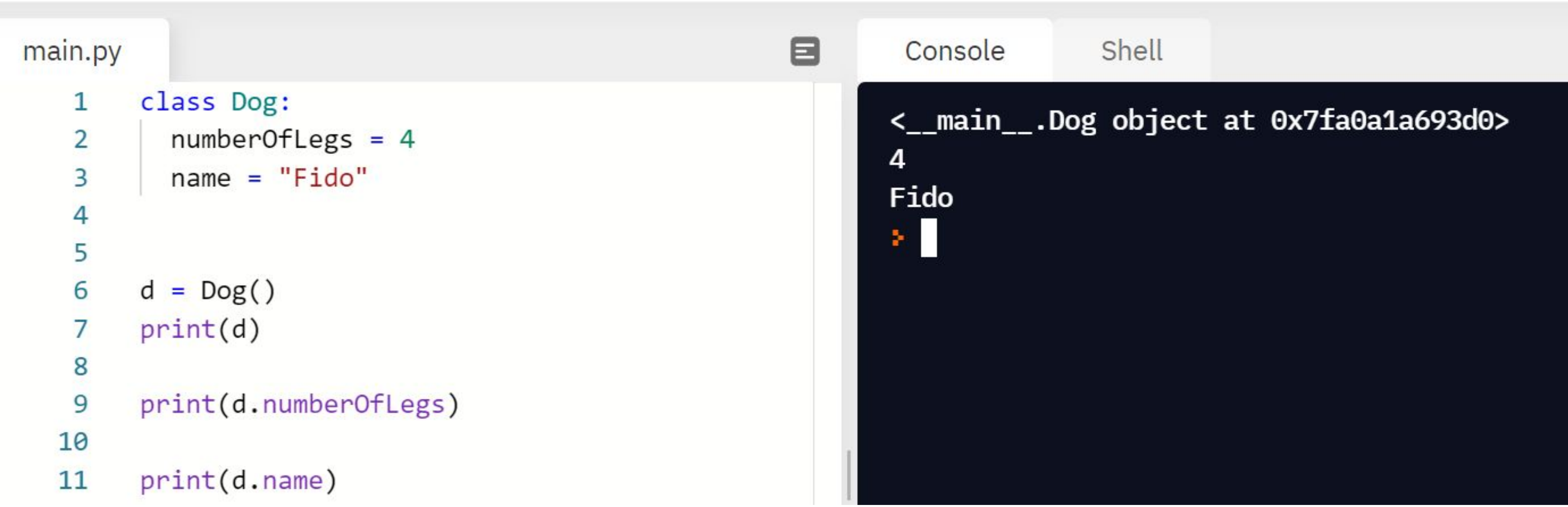
***An *instance* is simply an individual object from a certain class.**

Creating a class

You use the keyword ‘class’ to tell Python you are about to define a new class. Each class must have a name, and it should be descriptive for what the class’s purpose is. Every class has statements, attributes, and perhaps functions that belong to it.

```
class name:
    statements
```

Examples

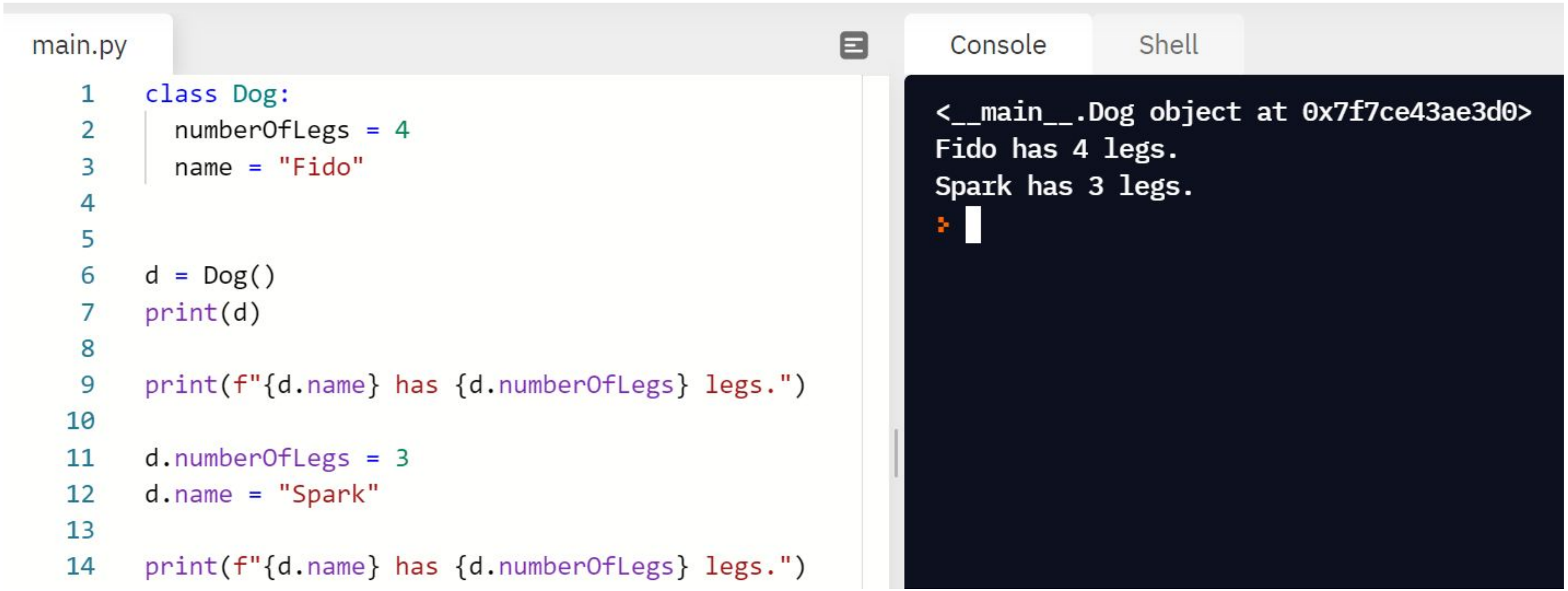


```python
class Dog:
  numberOfLegs = 4
  name = "Fido"


d = Dog()
print(d)

print(d.numberOfLegs)

print(d.name)
```

```
<__main__.Dog object at 0x7fa0a1a693d0>
4
Fido
>
```

What’s happening here? (side note: ‘numberOfLegs’ and ‘name’ are considered the *fields* or *attributes* of our class)

Examples

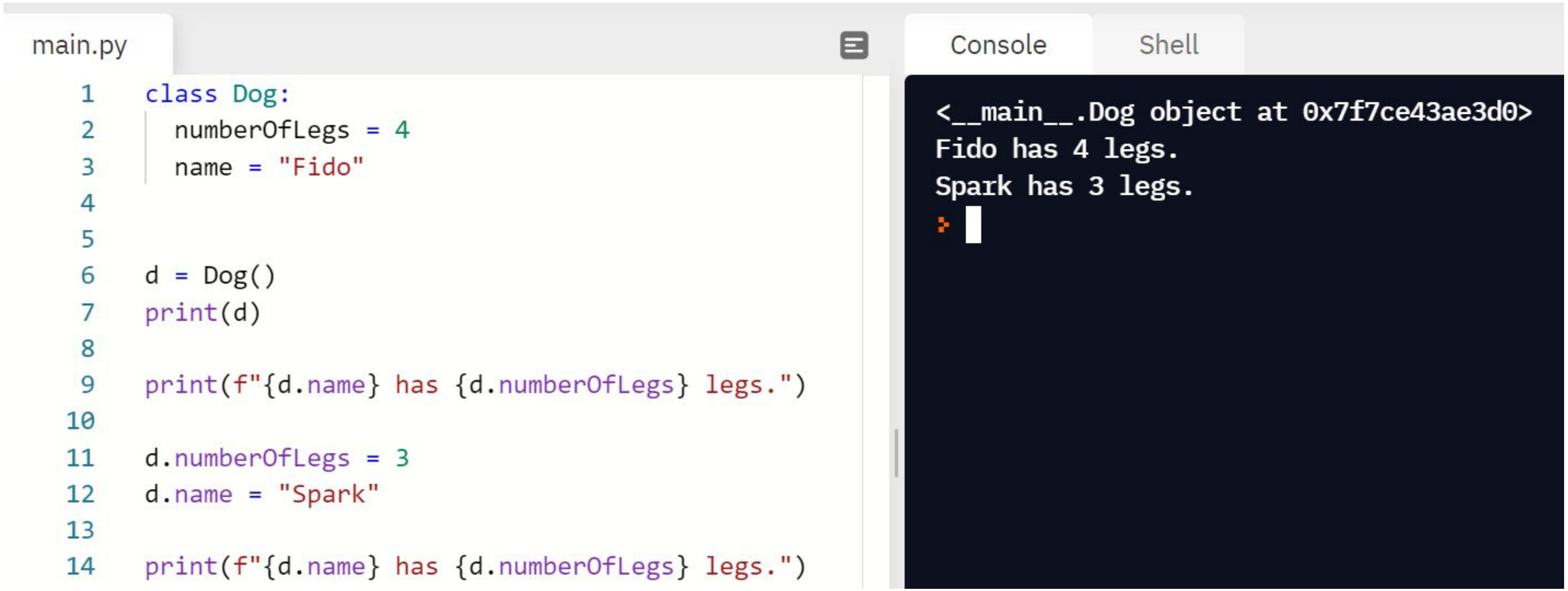


```python
1   class Dog:
2     numberOfLegs = 4
3     name = "Fido"
4
5
6   d = Dog()
7   print(d)
8
9   print(f"{d.name} has {d.numberOfLegs} legs.")
10
11  d.numberOfLegs = 3
12  d.name = "Spark"
13
14  print(f"{d.name} has {d.numberOfLegs} legs.")
```

```
<__main__.Dog object at 0x7f7ce43ae3d0>
Fido has 4 legs.
Spark has 3 legs.
>
```

What’s happening in lines 11 and 12? Is our Dog object mutable or immutable?

What if I wanted the dog Fido AND the dog Spark?

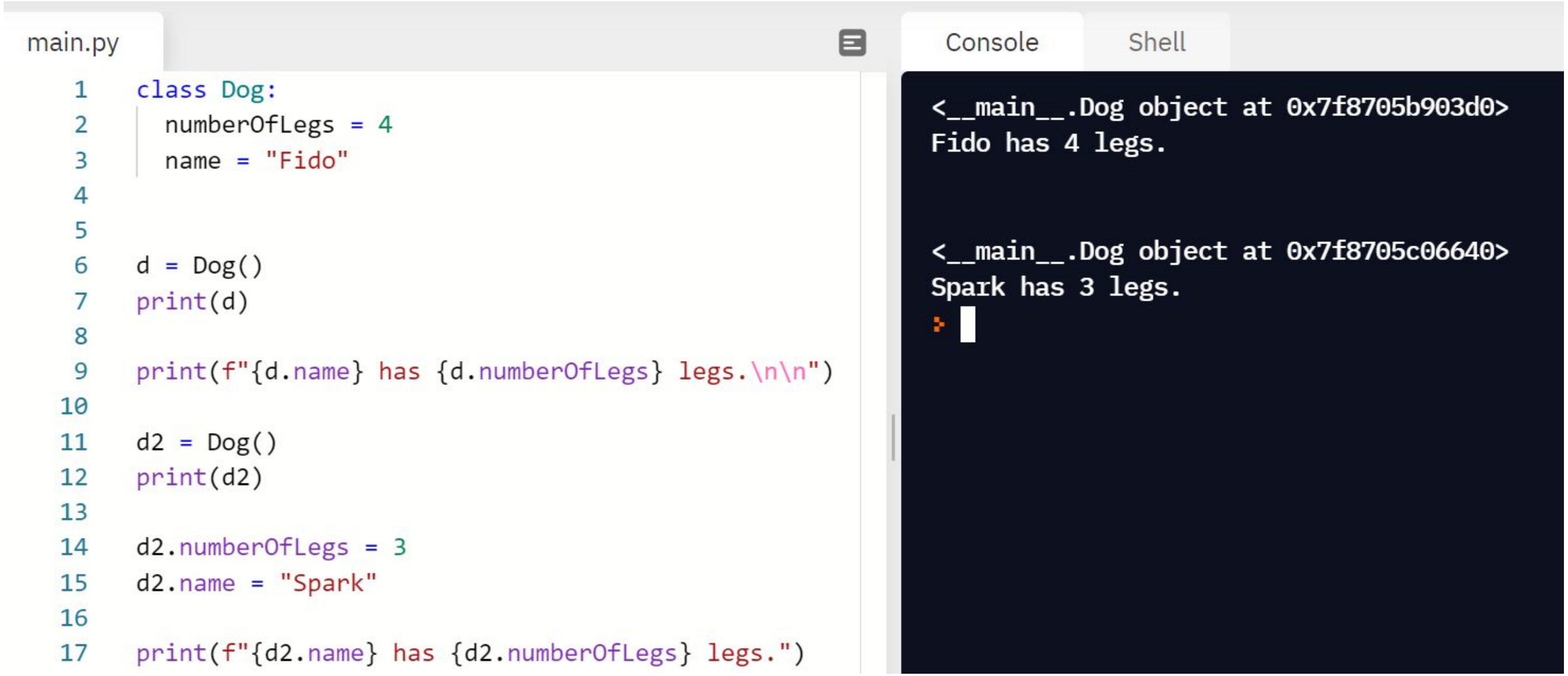


```python
class Dog:
  numberOfLegs = 4
  name = "Fido"


d = Dog()
print(d)

print(f"{d.name} has {d.numberOfLegs} legs.")

d.numberOfLegs = 3
d.name = "Spark"

print(f"{d.name} has {d.numberOfLegs} legs.")
```

```
<__main__.Dog object at 0x7f7ce43ae3d0>
Fido has 4 legs.
Spark has 3 legs.
>
```

What could I do in my code?

What if I wanted the dog Fido AND the dog Spark?



```python
class Dog:
  numberOfLegs = 4
  name = "Fido"


d = Dog()
print(d)

print(f"{d.name} has {d.numberOfLegs} legs.\n\n")

d2 = Dog()
print(d2)

d2.numberOfLegs = 3
d2.name = "Spark"

print(f"{d2.name} has {d2.numberOfLegs} legs.")
```

```
<__main__.Dog object at 0x7f8705b903d0>
Fido has 4 legs.


<__main__.Dog object at 0x7f8705c06640>
Spark has 3 legs.
```

I created a new Dog object. Notice how this object has its own place in memory.

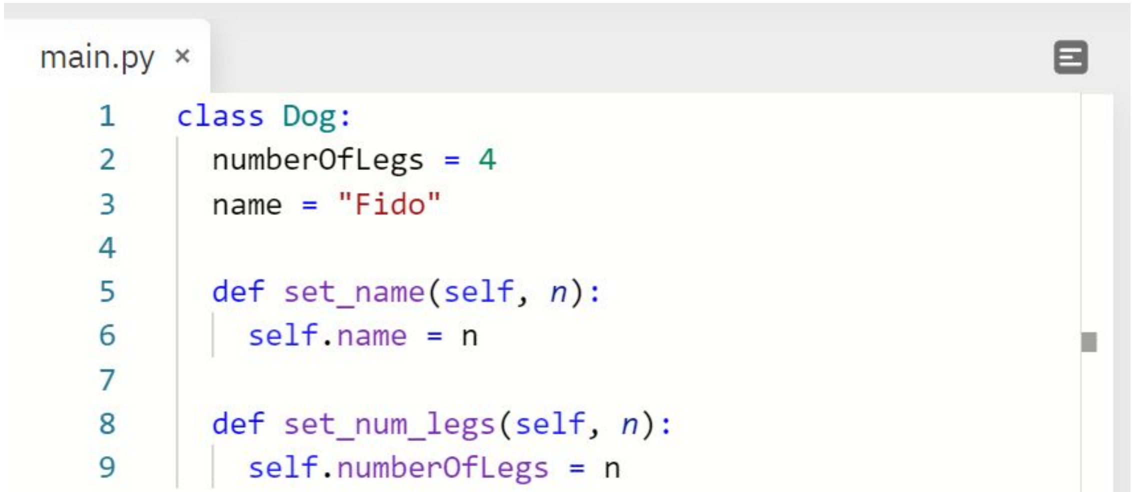
Classes

Since my class is a *blueprint*, I might want some of the data belonging to my class to be entered by the user, or by the program.

For example, maybe I want to make my **Dog** class have a user-specified dog's name and dog's #of legs.

Look at how we use "self" here:

(side note: **set_name** and **set_num_legs** are considered *methods* in my class)

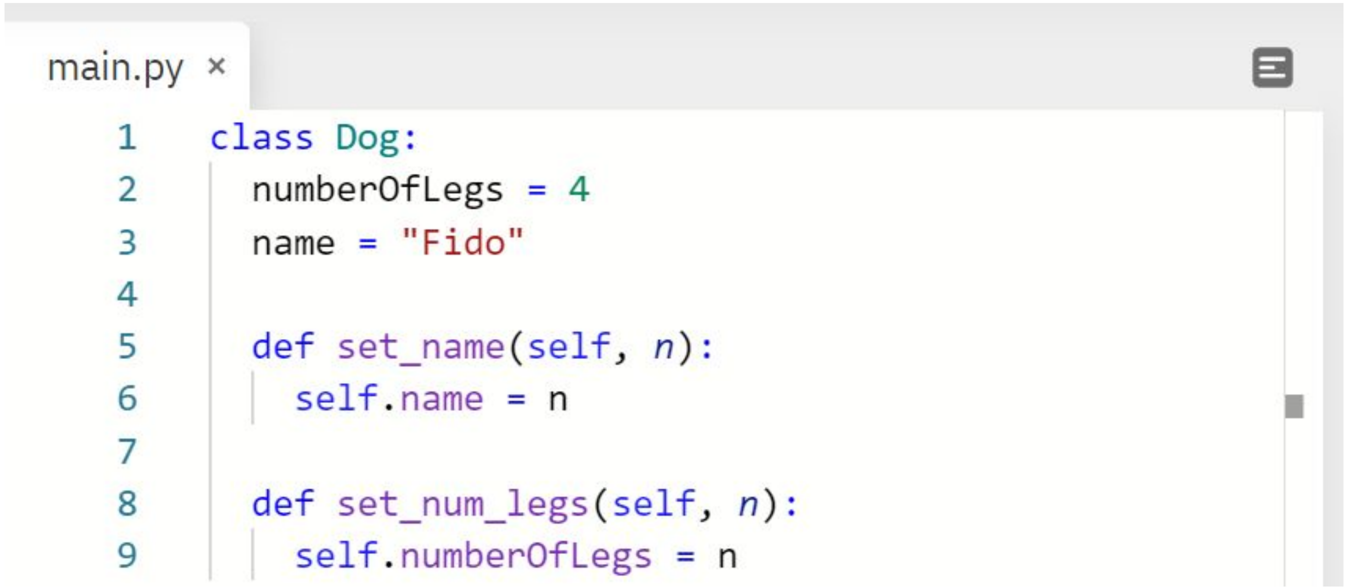


main.py

```
class Dog:
  numberOfLegs = 4
  name = "Fido"

  def set_name(self, n):
    self.name = n

  def set_num_legs(self, n):
    self.numberOfLegs = n
```

Classes

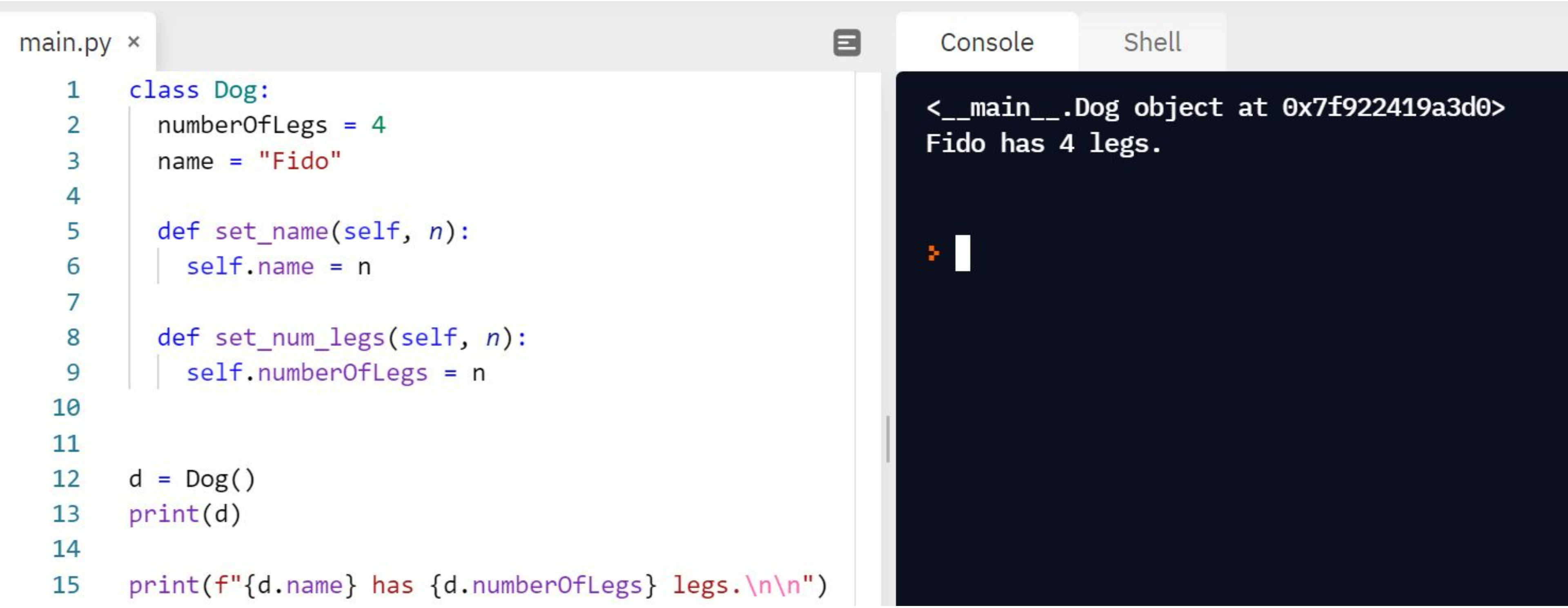
Given this code, whenever I create an *instance* of my Dog class (aka a new Dog object), what will the object's name and numberOfLegs be?



main.py ×

```python
class Dog:
  numberOfLegs = 4
  name = "Fido"

  def set_name(self, n):
    self.name = n

  def set_num_legs(self, n):
    self.numberOfLegs = n
```

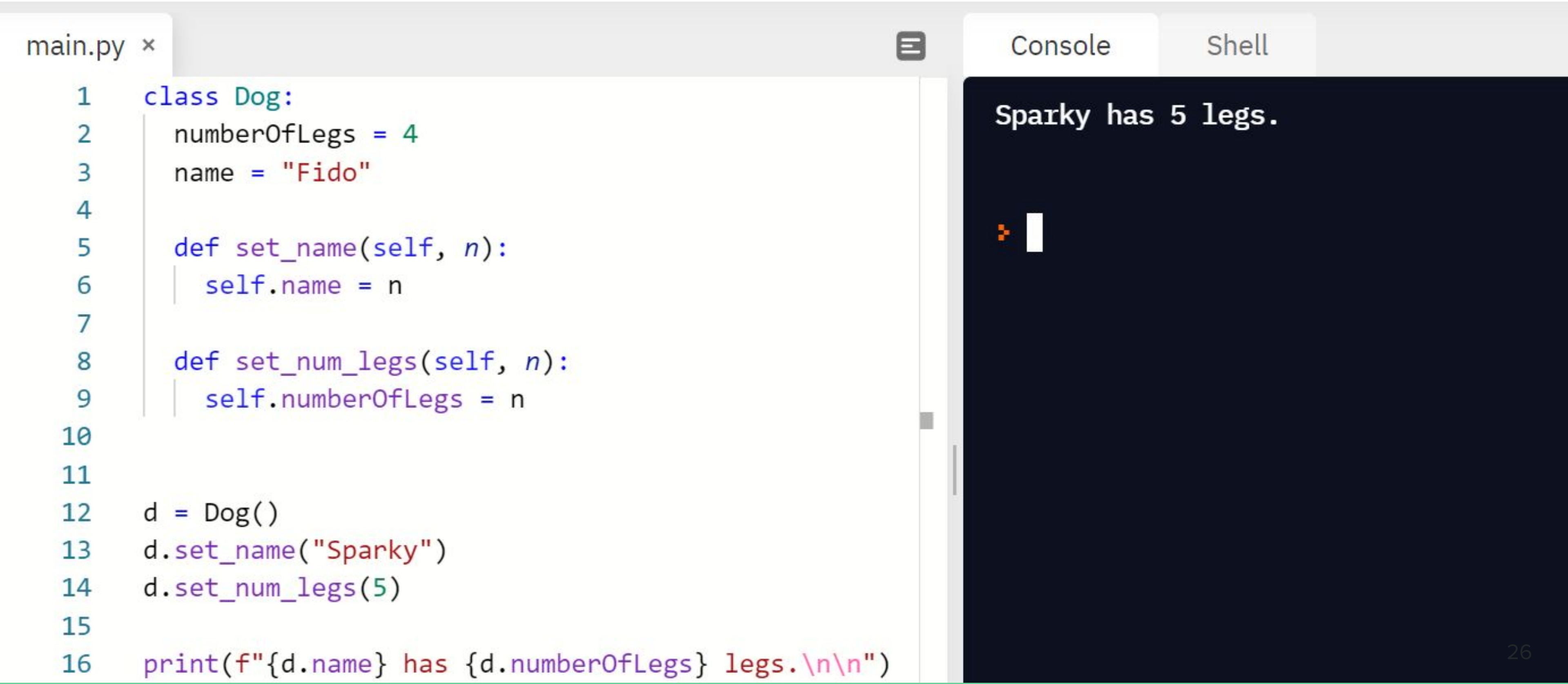
Creating an instance of the Dog class



```python
class Dog:
  numberOfLegs = 4
  name = "Fido"

  def set_name(self, n):
    self.name = n

  def set_num_legs(self, n):
    self.numberOfLegs = n


d = Dog()
print(d)

print(f"{d.name} has {d.numberOfLegs} legs.\n\n")
```

```
<__main__.Dog object at 0x7f922419a3d0>
Fido has 4 legs.
```

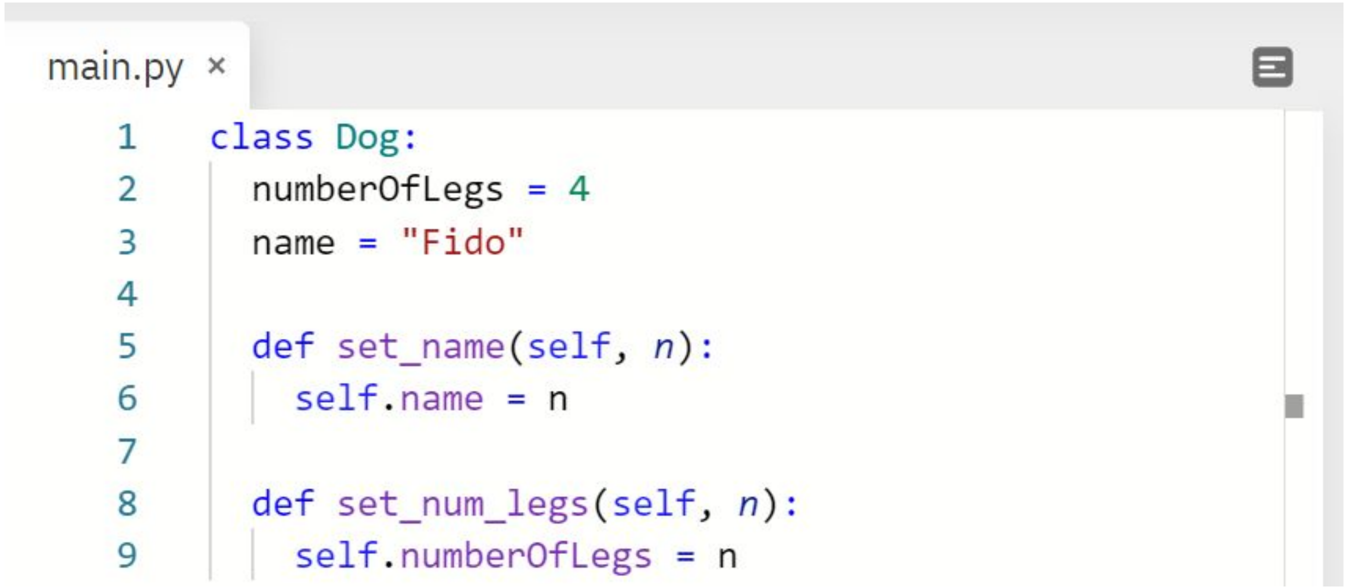
What’s different here?



```python
class Dog:
  numberOfLegs = 4
  name = "Fido"

  def set_name(self, n):
    self.name = n

  def set_num_legs(self, n):
    self.numberOfLegs = n


d = Dog()
d.set_name("Sparky")
d.set_num_legs(5)

print(f"{d.name} has {d.numberOfLegs} legs.\n\n")
```

```
Sparky has 5 legs.
```

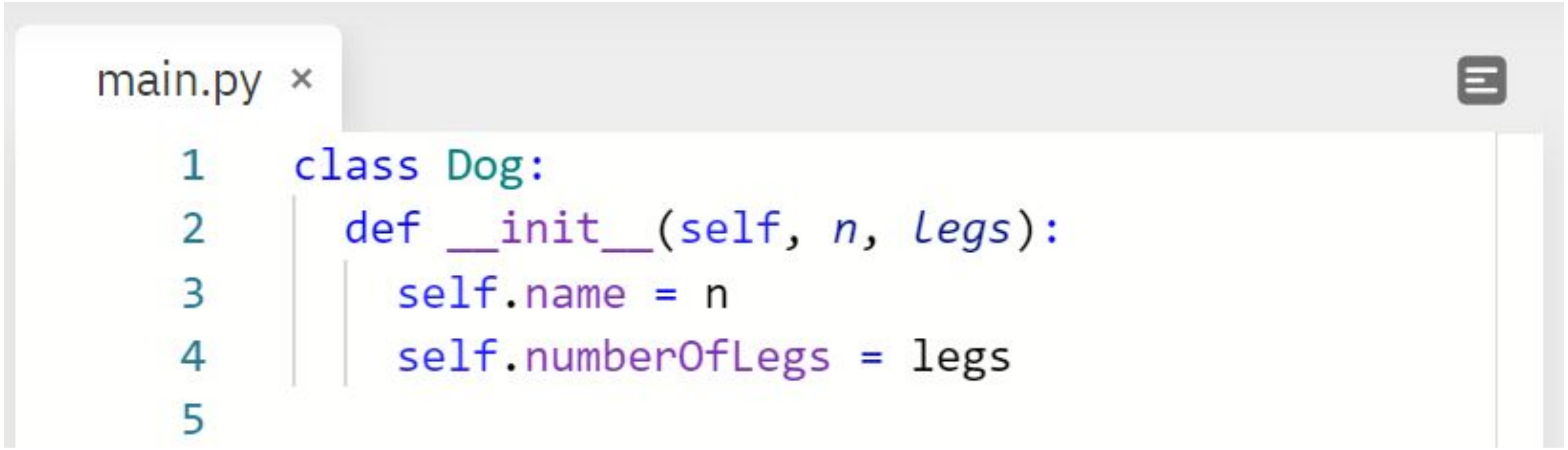
What’s different here?


main.py ×

```python
class Dog:
  numberOfLegs = 4
  name = "Fido"

  def set_name(self, n):
    self.name = n

  def set_num_legs(self, n):
    self.numberOfLegs = n
```

Classes

Now, we could also change our Dog class to TAKE IN a dog’s name and # of legs when we create a new instance of it:

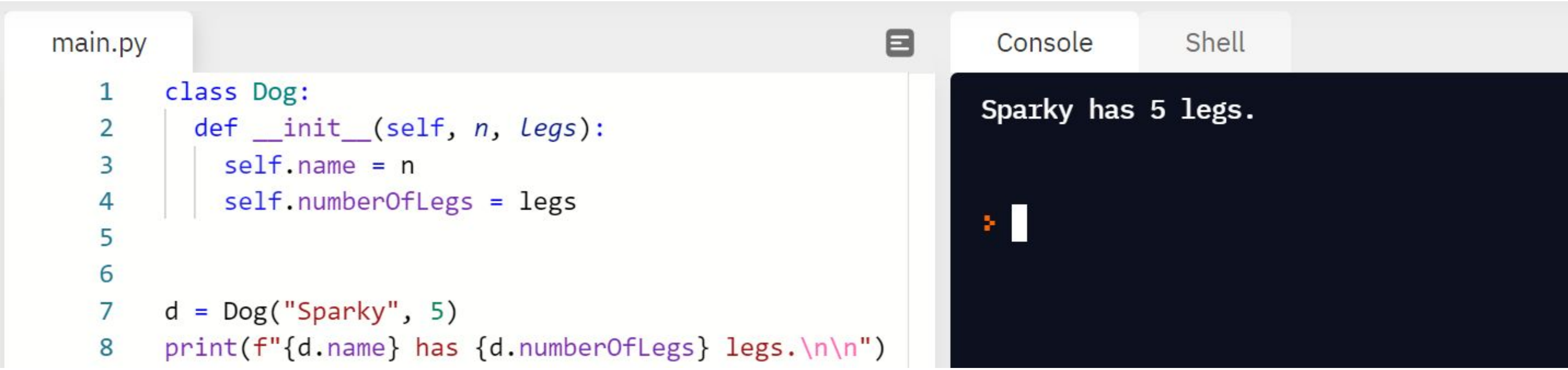


main.py ×

```python
class Dog:
  def __init__(self, n, legs):
    self.name = n
    self.numberOfLegs = legs

```

We can modify our class’s constructor (the way our class gets created / instantiated) using the **__init__** built-in method of our class. You always need to pass *self* in the constructor!

Creating an instance of our new Dog class



```python
class Dog:
  def __init__(self, n, legs):
    self.name = n
    self.numberOfLegs = legs


d = Dog("Sparky", 5)
print(f"{d.name} has {d.numberOfLegs} legs.\n\n")
```

```
Sparky has 5 legs.


>
```

What's different?

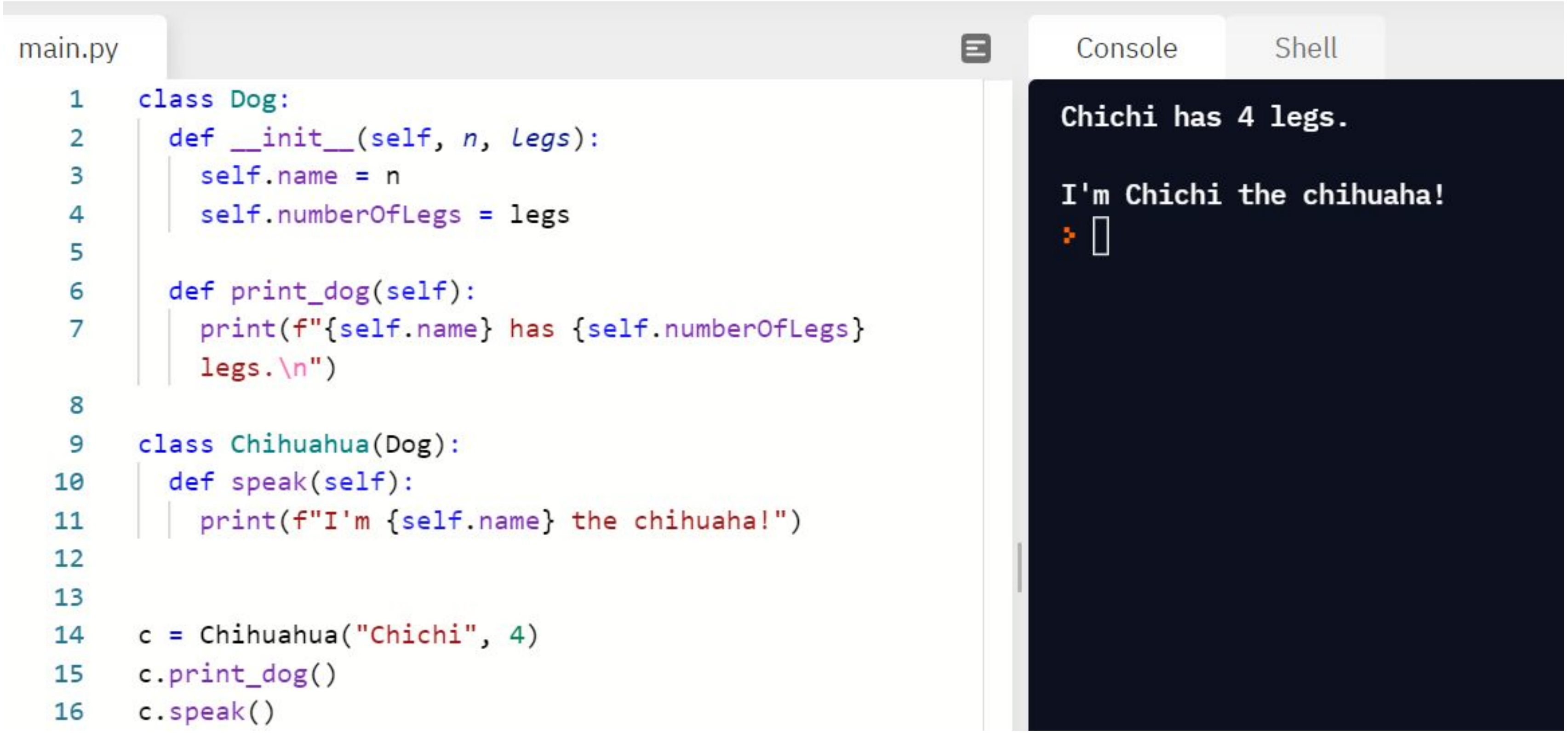
Inheritance

While I think inheritance is out of the scope of the Coding 101 program, I'd still like to mention it to you all so you at least know what it is.

Inheritance allows us to define a class that inherits all the methods and properties from another class. What does that mean?

By using inheritance, we can basically have a "subclass" or "child" of another class (often referred to as the "parent" class), such that our subclass/child gets to access everything the parent class has access to, such as its methods, fields, etc.

Inheritance example



main.py

```python
class Dog:
  def __init__(self, n, legs):
    self.name = n
    self.numberOfLegs = legs

  def print_dog(self):
    print(f"{self.name} has {self.numberOfLegs}
    legs.\n")

class Chihuahua(Dog):
  def speak(self):
    print(f"I'm {self.name} the chihuaha!")


c = Chihuahua("Chichi", 4)
c.print_dog()
c.speak()
```

```
Chichi has 4 legs.

I'm Chichi the chihuaha!
> 
```

Live coding