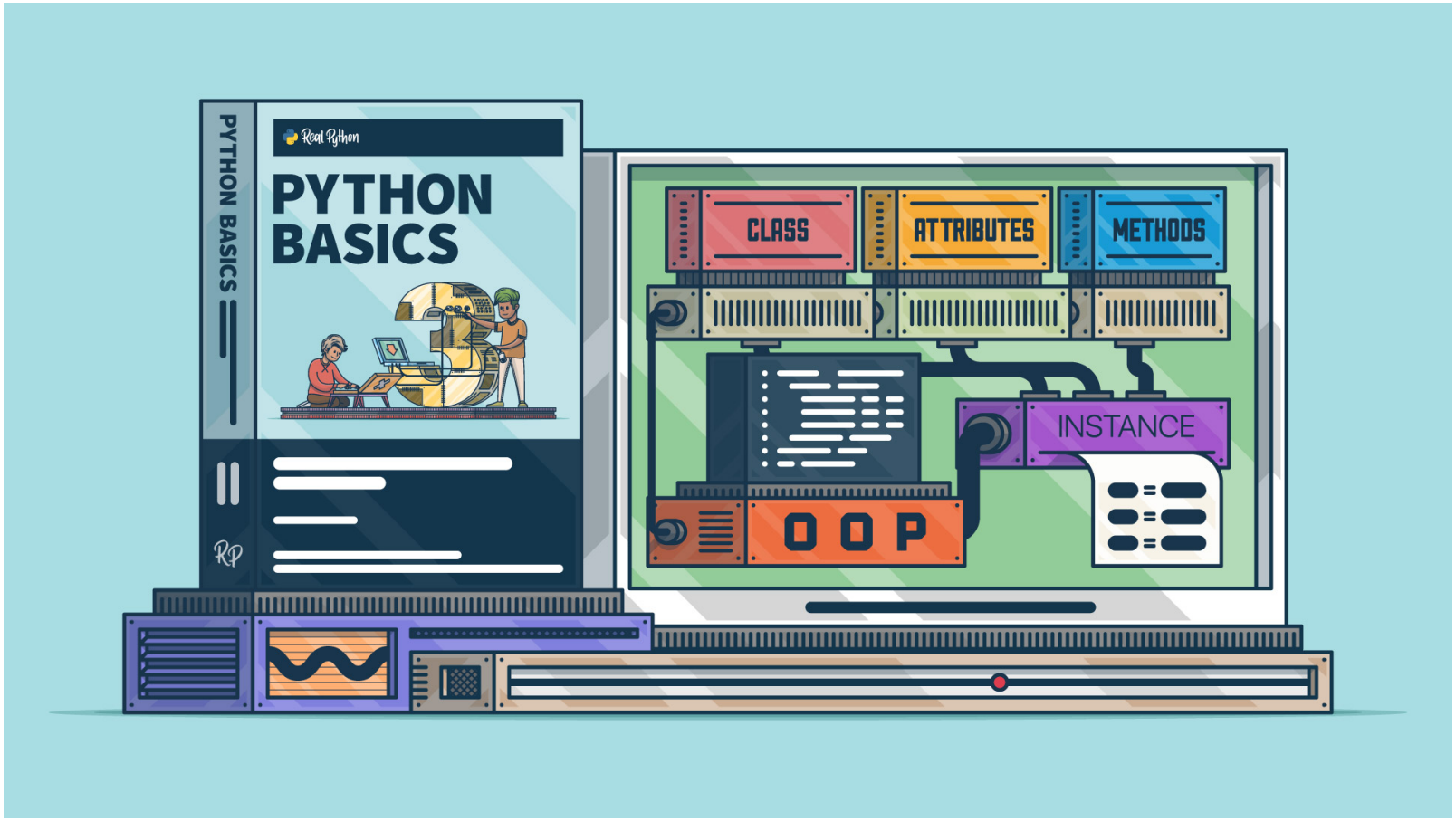
Python Basics: Object Oriented Programming



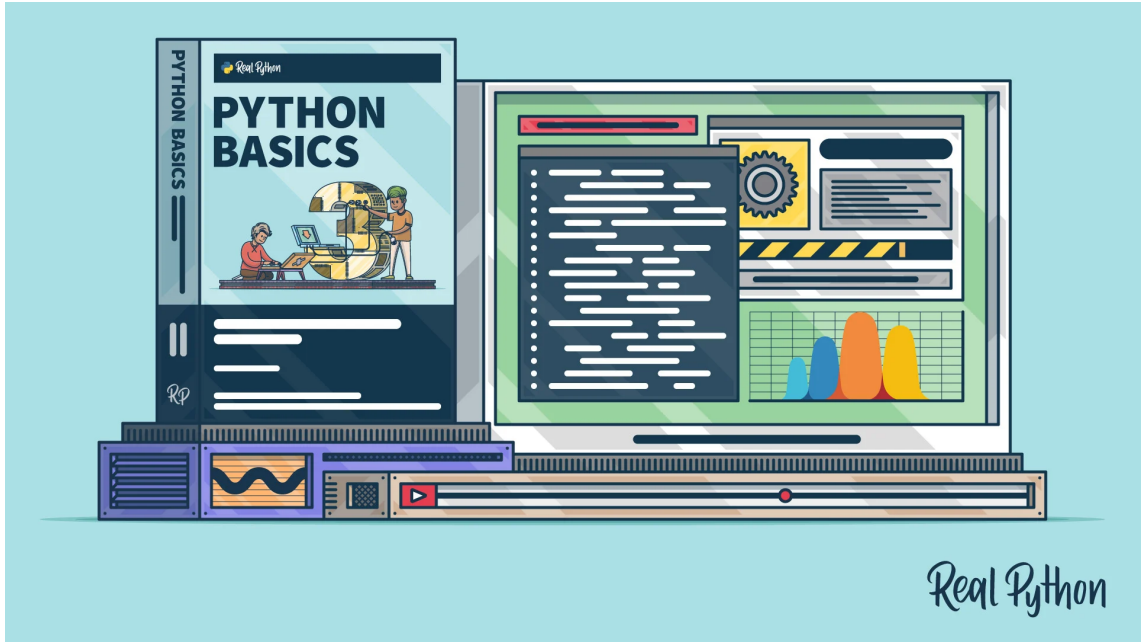
Python Basics: Object Oriented Programming

What you'll need:

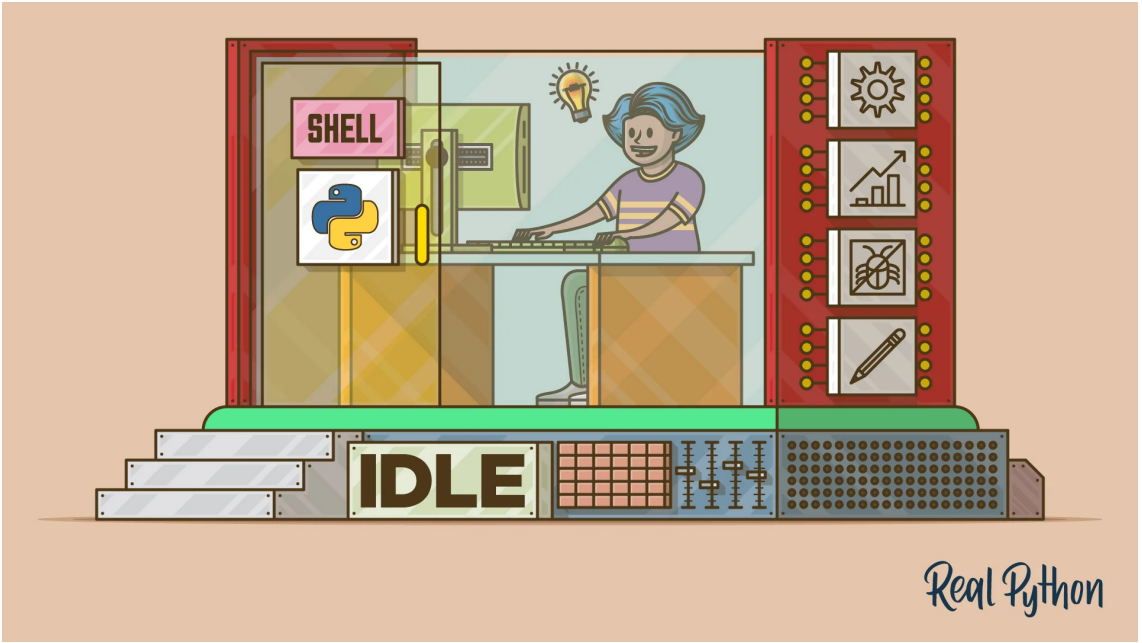
- **IDLE: I**ntegrated **D**evelopment and **L**earning **E**nvironment

Python Basics: Object Oriented Programming

What you'll need:



Python Basics: Setting Up Python



Starting With Python IDLE

Python Basics: Object Oriented Programming

In this course you'll:

- Get an idea of what Object Oriented Programming is all about
- Start to understand why OOP is useful
- Understand the jargon around OOP in Python
- Create classes and instances of classes
- Understand instance constructors
- Understand class and instance attributes
- Manipulate and compare instances
- Create and use methods and special methods

Introducing Object Oriented Programming

- **What is Object Oriented Programming?**
  - It’s a style of programming
  - It involves modelling things or concepts
  - It’s a way to group related functionality
  - It’s a tool to structure your code
  - It’s a skill to practice

Understanding Why You Need Classes

- How to store data?
- Where to write code?
- How to operate on data?
- Where to put the code that operates on data?

Understanding Why You Need Classes

- You Need Classes to Make Your Life as a Developer Easier

Understanding Why You Need Classes

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

Understanding Why You Need Classes

```
kirk[1]
spock[1]
mccoy[1]
```

Understanding Why You Need Classes

```
kirk[1]  # 34
spock[1]  # 35
mccoy[1]  # "Chief Medical Officer"
```

Understanding Why You Need Classes

```
kirk.age  # 34
spock.age  # 35
mccoy.age  # AttributeError
```

Understanding Why You Need Classes

```
x1 = 1
y1 = 2

point = (1, 2)
```

Understanding Why You Need Classes

```
point[0] = 1
point[1] = 2
```

Understanding Why You Need Classes

```
point.x = 1
point.y = 2
```

Understanding Why You Need Classes

A convenient structure to:

- Store and operate on data
- Encapsulate code

Understanding Why You Need Classes

You Need Classes to Make Your Life as a Developer Easier

Disambiguating Class and Instance

- Class
  - Blueprint
  - Prototype
  - Model
- Instance
  - Specimen
  - Occurence
  - Occasion

Disambiguating Class and Instance

Class	Instance
The mammal *Canis familiaris*	Your dog Spike
The concept of a point in two-dimensional space	The center of a canvas
The idea of a chair	The actual chair you are sitting on

Disambiguating Class and Instance

- A class is like a blueprint
- An instance is like something made from the blueprint

Disambiguating Class and Instance

Analogies are limited!

Creating a Class

```
class Point:
    pass

class Doggo:
    pass
```

Creating a Class

Naming Convention

- CapitalizedWords
- CamelCase
- PascalCase

Instantiating an Object

```
>>> class Point:
...     pass
...
>>> Point()
<__main__.Point object at 0x106702d30>
>>> Point()
<__main__.Point object at 0x0004ccc90>
```

Understanding the Constructor and Attributes

- The `.__init__()` constructor
- Attributes
- The `self` parameter

Understanding the Constructor and Attributes

```
class Point:
    ...
```

Understanding the Constructor and Attributes

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

Understanding the Constructor and Attributes

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

class Doggo:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Understanding the Constructor and Attributes

What is `self`?

- `self` refers to the *instance* of the class
- On “its *self*”
- A name commonly given to the first argument passed to the constructor
- A convention to call it `self` and you can call it something else
- Don’t call it something else

Understanding the Constructor and Attributes

- You call the constructor when you instantiate a class
- You define a `self` parameter to the constructor
- You can assign attribute values to the `self` object
- You can put any instance setup in the constructor

Understanding Class and Instance Attributes

```
class Doggo:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Understanding Class and Instance Attributes

```
class Doggo:
    species = "Canis familiaris" #  class attribute
    def __init__(self, name, age):
        self.name = name  # instance attribute
        self.age = age  # instance attribute
```

Understanding Class and Instance Attributes

```
class Point:
    dimensions = 2
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Understanding Class and Instance Attributes

```
class Point:
    dimensions = 2  # class attribute
    def __init__(self, x, y):
        self.x = x  # instance attribute
        self.y = y  # instance attribute
```

Understanding Class and Instance Attributes

- An instance attribute's value is specific to a particular instance.
- A class attribute is available from the class itself, and all instances derived from that class.

Instantiating With Attributes

```
class Point:
    dimensions = 2
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Instantiating With Attributes

```
>>> Point()
```

Instantiating With Attributes

```
>>> Point()
Traceback (most recent call last):
    ...
TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'
```

Instantiating With Attributes

```
>>> Point()
Traceback (most recent call last):
    ...
TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'

>>> origin = Point(0, 0)

>>> target = Point(10, 15)
```

Instantiating With Attributes

```
>>> origin = Point(0, 0)
>>> target = Point(10, 15)
```

Instantiating With Attributes

```
>>> origin = Point(0, 0)
>>> target = Point(10, 15)

>>> origin.x
0
>>> target.y
15
```

Instantiating With Attributes

```
>>> origin = Point(0, 0)
>>> target = Point(10, 15)

>>> origin.x
0
>>> target.y
15

>>> Point.dimensions
2
>>> origin.dimensions
2
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
2
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
2

>>> Point.dimensions = 3
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
2

>>> Point.dimensions = 3
>>> Point.dimensions
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
2

>>> Point.dimensions = 3
>>> Point.dimensions
3
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
2

>>> Point.dimensions = 3
>>> Point.dimensions
3
>>> target.dimensions
```

Instantiating With Attributes

You can change attributes at runtime:

```
>>> target.x = 12
>>> target.x
12
>>> target.dimensions
2

>>> Point.dimensions = 3
>>> Point.dimensions
3
>>> target.dimensions
3
```

Instantiating With Attributes

- If you have a constructor:
  - You need to pass all the arguments apart from `self`
  - Or else you'll get an error
- If the constructor assigns instance attributes:
  - You can access them from the instance
  - You can change them at runtime
- If your class has class attributes:
  - You can access them from the class or instance
  - You can change them at runtime

Checking Equivalence Between Objects

- `isinstance()` - recommended
- `type()`

Checking Equivalence Between Objects

```
>>> origin = Point(0, 0)
>>> target = Point(0, 0)

>>> origin == target
False
>>> origin is target
False
```

Checking Equivalence Between Objects

```
>>> origin = Point(0, 0)
>>> target = Point(0, 0)

>>> origin == target
False
>>> origin is target
False

>>> isinstance(origin, Point)
True
>>> isinstance(target, Point)
True

>>> type(origin) == type(target)
True
```

Introducing Instance Methods

Functions that are attached to instances

Introducing Instance Methods

```
class Doggo:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Introducing Instance Methods

```
class Doggo:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old, and is a good doggo."

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Introducing Instance Methods

```
>>> miles = Doggo("Miles", 4)

>>> miles.description()
'Miles is 4 years old, and is a good doggo.'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

Introducing Dunder Methods

Also called **special instance methods**

Introducing Dunder Methods

What are dunder methods used for?

A problem:

```
>>> names = ["David", "Dan", "Joanna", "Fletcher"]
>>> print(names)
['David', 'Dan', 'Joanna', 'Fletcher']

>>> print(miles)
```

Introducing Dunder Methods

What are dunder methods used for?

A problem:

```
>>> names = ["David", "Dan", "Joanna", "Fletcher"]
>>> print(names)
['David', 'Dan', 'Joanna', 'Fletcher']

>>> print(miles)
<__main__.Doggo object at 0x00aeff70> 🤮
```

Introducing Dunder Methods

```
class Doggo:
    species = "Canis familiaris"
    def __init__(self, name, age): ...
    def description(self): ...
    def speak(self, sound): ...

    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Introducing Dunder Methods

```
>>> miles = Doggo("Miles", 4)
>>> print(miles)
```

Introducing Dunder Methods

```
>>> miles = Doggo("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

Introducing Dunder Methods

```
class Point():
    def __init__(x, y): ...

    def __add__(self, other):
        self.x = self.x + other.x
        self.y = self.y + other.y

    def __str__(self):
        return f"Point at x:{self.x}, y: {self.y}"
```

Introducing Dunder Methods

```
>>> center = Point(50, 50)
>>> print(center)
```

Introducing Dunder Methods

```
>>> center = Point(50, 50)
>>> print(center)
Point at x:50, y: 50
```

Introducing Dunder Methods

```
>>> center = Point(50, 50)
>>> print(center)
Point at x:50, y: 50

>>> distance = Point(25, 25)

>>> center + distance
```

Introducing Dunder Methods

```
>>> center = Point(50, 50)
>>> print(center)
Point at x:50, y: 50

>>> distance = Point(25, 25)

>>> center + distance

>>> print(center)
Point at x:75, y: 75
```

Introducing Dunder Methods

- *Dunder* is jargon for *double underscore*
- Special instance methods can be used to override basic behavior
  - Representation as a string
  - Behavior with operators such as `+`
- There are many variations, check out the documentation

Seeing Dunder Methods in Action

Exercising Your New Knowledge

- Modify the `Doggo` class to include another attribute `coat_color`, and add it to the `.__init__()` constructor.
- Create a `Car` class with a few instances that are appropriate for cars.
- Add a `.mileage` attribute to the `Car` class and add a `.drive()` instance method to the class which will add a number of miles to the mileage.

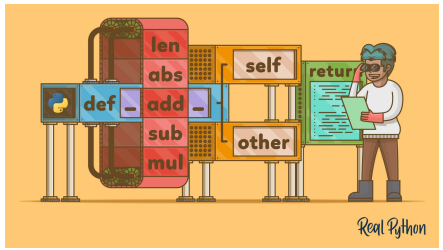
Python Basics: Object Oriented Programming

- Learned about the object-oriented programming paradigm
- Defined a class
- Added attributes
- Added methods
- Used objects created from classes

Additional Resources

Object-Oriented Programming (OOP) in Python 3

Getters and Setters: Manage Attributes in Python



Operator and Function Overloading in Custom Python Classes

Python Basics: Object Oriented Programming