

00010001

CHAPTER 17

Introduction To Classes & Object-Oriented Programming

Ch-17: Introduction To Classes & Object-Oriented Programming

Learning Objectives

- *State the purpose of classes*
- *Use UML class diagrams to communicate application design*
- *Use the class keyword to define classes*
- *State the purpose of the `__new__` and `__init__` special methods*
- *Explain what it means to create an instance of a class*
- *State the purpose of the `__self__` attribute*
- *State the difference between functions and methods*
- *Create instance variables*
- *Define methods using the `def` keyword*
- *Call instance methods using dot notation*
- *Define properties using the `@property` decorator*
- *Define property setters*

0
0
0
1
0
0
0
1

INTRODUCTION

Classes are a foundational concept in object-oriented analysis, design, and programming. Classes allow you to gain a clear mental understanding of how your code is organized and structured, leading to clean application architectures. Clean application architectures lead to code that's easier to manage, maintain, and evolve.

Classes enable you to co-locate data along with the methods required to process that data into a single, coherent user-defined data type. Designing an application with classes allows you to think in terms of objects and the interactions between them. This change in thinking helps you to tame *conceptual complexity*.

Classes, along with code modules, allow you to tame *physical complexity* as well. Physical complexity refers to the number of source code files and other artifacts a project contains. As a general rule: the bigger the application — the greater its physical complexity.

In this chapter, you will learn how to define and use classes in your programs. Along the way I will show you how to visualize classes and their relationships between each other with the help of Unified Modeling Language (UML), which is a standardized modeling language used in software engineering to design, model, and document object-oriented software systems. UML provides a set of graphical notations for representing various system aspects including structure, behavior, and interaction.

1 CLASSES 101

In this section, you will learn what classes are, how to define them, and how to use them in your programs.

1.1 WHAT IS A CLASS?

In the world of software engineering, the term *class* has several meanings depending upon the context in which it appears of which there are three we are concerned with: object-oriented *analysis*, object-oriented *design*, and object-oriented *programming*. I will treat object-oriented analysis and design (OOA&D) as one context, as the lines between the tasks performed within these two activities are often blurred.

In the context of object-oriented *analysis & design*, a class is used to represent the concept or notion of an entity within a problem domain. There is usually a one-to-one mapping from real-world objects in the problem domain to classes within the design domain. Following the problem domain analysis, an initial software design is created which attempts to identify the properties and methods required by each class. Such a design can be expressed visually using Unified Modeling Language (UML) class diagrams. I talk more about UML later in the chapter. Note that the activities referred to here as *analysis* and *design* are performed iteratively in an ongoing effort to better understand the problem domain and refine the design.

In the context of object-oriented *programming*, a class refers to a programming language construct that enables a programmer to implement in code the conceptual notion of a class as discovered during analysis and design. Python provides the `class` keyword, which lets you define *user-defined types* which serve as templates for the creation of objects within your program. A class

definition can specify what type of *data* an object can contain, what type of *behavior* an object can support, or both. Usually, it's both.

1.2 WHAT IS AN OBJECT?

A class is used to create one or more *objects* within your program. A class definition is just that, a definition. To actually do something with the class you must create an *instance* of the class. (You will also hear and see related terms such as *object instantiation*.) This results in an object of the class type being instantiated or created in memory. An object is nothing more than an area in memory that contains the data associated with a particular instance.

1.3 DEFINING A CLASS

OK, let's say you are working on a program that needs to represent the concept of a person with a few basic attributes like first name, middle name, and last name. Example 17.1 gives the listing for a Person class.

17.1 *person.py*

```

1  """Contains the definition of the Person class."""
2
3  class Person:
4      """Defines a Person class."""
5
6      # This is a class-wide attribute
7      # shared by all Person objects
8      count = 0
9
10     def __new__(cls, *args, **kwargs):
11         """Creates a new Person object."""
12         if __debug__:
13             print('__new__() method called...Person object created!')
14         instance = super().__new__(cls)
15         return instance
16
17
18     def __init__(self, first_name:str='John',
19                 middle_name:str='J', last_name:str='Doe')->None:
20         """Initializes Person object with known state."""
21         self.first_name = first_name
22         self.middle_name = middle_name
23         self.last_name = last_name
24         Person.count += 1
25         if __debug__:
26             print(f'__init__() method called...Person object initialized!')
27
28
29     def __str__(self)->str:
30         """Returns a string representation of the object."""
31         return f'{self.first_name} {self.middle_name} {self.last_name}'
32

```

Referring to example 17.1 — This example contains the definition for a class named Person in a module named person (i.e., *person.py*). The `class` keyword is used on line 3 to define a class named Person. Note that the class name starts with a capital letter while the module name is in lower-case. (See [PEP 8 — Style Guide for Python Code for naming guidelines](#).)

The `Person` class consists of one class-wide attribute named `count`, three special methods `__new__()`, `__init__()`, and `__str__()`, and three instance attributes `self.first_name`, `self.middle_name`, and `self.last_name`. Let's discuss each of these in more detail starting with the class-wide attribute.

On line 8, I declare a class-wide attribute named `count` and initialize it to zero. A class-wide attribute is shared by all instances created by the class. In this context, the term `instance` is synonymous with `object`.

On line 10, I have defined a `__new__()` method. The `__new__()` method is responsible for creating an object of type `Person` in memory. The `__new__()` method is also referred to as the *constructor*. You usually do not need to define a `__new__()` method in your classes. I have done so here to illustrate the order of method calls during object construction and initialization.

The definition for the `__init__()` method begins on line 18. The purpose of the `__init__()` method is to *initialize* the newly-instantiated object into a *known state*. This is where you put definitions for any instance attributes required by objects of this type. The first parameter listed in the `__init__()` method is `self`. The parameter name `self` represents an instance. The `self` parameter is then used to define instance attributes as shown on lines 21 through 23. Note how the instance attributes are accessed via the parameter name `self` while the class-wide attribute `count` is accessed via the class name `Person` as shown on line 24. Also, in addition to the `self` parameter, I declared three additional parameters for the `__init__()` method: `first_name`, `middle_name`, and `last_name`, and provided default values for each parameter.

Finally, the definition for the `__str__()` method begins on line 29. The purpose of the `__str__()` method is to return a string representation of the object. Since I am dealing with objects of type `Person`, I've coded the `__str__()` method to return the `self.first_name`, `self.middle_name`, and `self.last_name` instance attributes.

1.4 INSTANTIATING OBJECTS

The `Person` class can now be used to create objects. Example 17.2 shows the `Person` class in action.

17.2 *main.py*

```

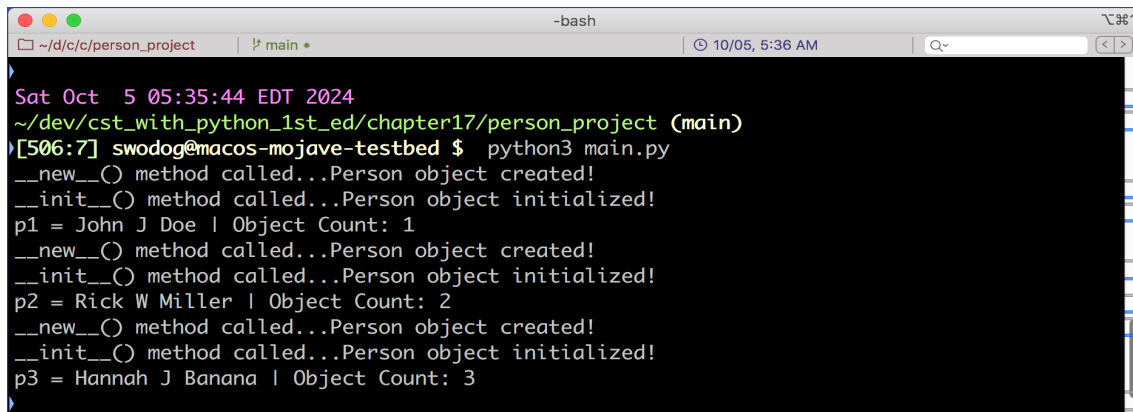
1  """Demonstrate object instantiation and attribute access."""
2
3  from person import Person
4
5  def main():
6      p1 = Person()
7      print(f'p1 = {p1} | Object Count: {Person.count}')
8      p2 = Person('Rick', 'W', 'Miller')
9      print(f'p2 = {p2} | Object Count: {Person.count}')
10     p3 = Person()
11     p3.first_name = 'Hannah'
12     p3.middle_name = 'J'
13     p3.last_name = 'Banana'
14     print(f'p3 = {p3} | Object Count: {Person.count}')
15
16
17     if __name__ == '__main__':
18         main()
19

```

Referring to example 17.2 — On line 3, I import the `Person` class from the `person` module. On line 6, I create an instance of a person object by *calling* the `Person` constructor with no arguments like so: `Person()`. The call to the `Person()` constructor returns a reference to the new object which is assigned to the variable `p1`. The variable `p1` is said to *reference* or *point* to a person object. On the next line, I print `p1` and the class-wide `Person.count` value.

On line 8, I create another instance of person only this time I am passing in arguments for the `first_name`, `middle_name`, and `last_name` parameters. The new object reference is assigned to the variable `p2` and on the following line, I print `p2` and the `Person.count` values to the console.

Finally, on line 10, I instantiate another person object, assign the reference to variable `p3`, then use `p3` to access and set each of the object's instance attributes directly. Figure 17-1 shows the results of running this program.



```

Sat Oct 5 05:35:44 EDT 2024
~/dev/cst_with_python_1st_ed/chapter17/person_project (main)
[506:7] swodog@macos-mojave-testbed $ python3 main.py
__new__() method called...Person object created!
__init__() method called...Person object initialized!
p1 = John J Doe | Object Count: 1
__new__() method called...Person object created!
__init__() method called...Person object initialized!
p2 = Rick W Miller | Object Count: 2
__new__() method called...Person object created!
__init__() method called...Person object initialized!
p3 = Hannah J Banana | Object Count: 3

```

Figure 17-1: Results of Running Example 17.2

Referring to figure 17-1 — Creating an instance of a class involves behind-the-scenes calls to the two special methods `__new__()` and `__init__()`. The `__new__()` method is called first which *creates* the object, followed by a call to the `__init__()` method, which *initializes* the newly-created object. You can see above that each call to the `Person()` constructor results in two messages being printed to the console — the first from the `__new__()` method and the second from the `__init__()` method.

1.5 TO `__NEW__()` OR NOT TO `__NEW__()`

You generally don't need to implement the `__new__()` method except in very special object-creation scenarios. You can read more about when to implement a `__new__()` method in the Python docs: https://docs.python.org/3/reference/datamodel.html#object.__new__



1.6 RULES TO PRESERVE YOUR SANITY

I'd like to offer a few rules to follow when defining and using classes. Adhering to this short list of rules will help to preserve your sanity as your project grows in conceptual and physical complexity. Several come straight from PEP 8 while the others come from hard lessons learned.

Rule	Explanation
One Class per Module	Place class definitions in separate modules. If you have multiple classes in your application, place each class definition in a dedicated module with the same name as the class. (i.e., The <code>person.py</code> module contains the <code>Person</code> class definition.)
Access Class-Wide Attributes via Class Name	Class-wide attributes are shared by all instances of the class. Prefix the attribute name with the class name. (i.e., <code>Person.count</code>)
Access Instance Attributes via Reference Name	Instance attribute values are unique to each object and can be accessed via the reference name. (i.e., <code>p1.first_name</code>)
Instantiate Objects in Known State	Implement an <code>__init__()</code> method to ensure objects are initialized into a known state. Provide default parameter values so the class constructor can be called without arguments.
Observe Good Module Naming Conventions	Module names should be lower-case with underscores separating each word of a multi-word module name. (a.k.a., <code>snake_case</code>)
Observe Good Class Naming Conventions	Class names should begin with an upper-case letter with the first letter of each subsequent word in multi-word class names capitalized. (a.k.a., <code>PascalCase</code>)

Table 17-1: Rules To Preserve Your Sanity When Defining and Using Classes

Referring to table 17-1 — The *One Class per Module* rule helps tame both conceptual and physical complexity. You should be able to locate a class definition by looking at the module name. If you define multiple classes in one module, locating a particular class becomes a lot more challenging. You should follow this rule even for small projects. When you're actively working on a project you have a good idea where all your code is located because your project layout is fresh in your mind. It's when you step away from your project for a while that you'll find upon your return the design and implementation decisions you made several months, weeks, or even just a few days ago have begun to fade from memory. If you put multiple class definitions in one module you'll have to go on a scavenger hunt to find the class you're looking for. **Logical artifact naming is a powerful organizational tool.**

1.7 ENOUGH ROPE TO BLOW YOUR LEG CLEAN OFF

I have a wonderful little volume on my bookshelf titled: *Enough Rope To Shoot Yourself In The Foot: Rules for C and C++ Programming*, by Allen I. Holub. It's a must read for anyone learning C or C++ and is absolutely packed with great advice. Essentially what it says is: *Just because a programming language allows you to do something doesn't mean you should.*

Python is a powerful and flexible programming language and like C and C++, you can do things in Python that can lead to disaster if you're not fully aware of the consequences. I'm going to make a slight revision to example 17.2 to illustrate my point.

```

1  """Demonstrate object instantiation and attribute access."""
2
3  from person import Person
4
5  def main():
6      p1 = Person()
7      print(f'p1 = {p1} | Object Count: {p1.count}')
8      p2 = Person('Rick', 'W', 'Miller')
9      print(f'p2 = {p2} | Object Count: {p2.count}')
10     p3 = Person()
11     p3.first_name = 'Hannah'
12     p3.middle_name = 'J'
13     p3.last_name = 'Banana'
14     print(f'p3 = {p3} | Object Count: {p3.count}')
15
16     print('*' * 40)
17
18     # A rookie mistake
19     p1.count = 100
20     print(f'p1 = {p1} | Object Count: {p1.count}')
21     print(f'p2 = {p2} | Object Count: {p2.count}')
22     print(f'p3 = {p3} | Object Count: {p3.count}')
23
24 if __name__ == '__main__':
25     main()
26

```

Referring to example 17.3 — On lines 7, 9, and 14, I am now *reading* the class-wide count attribute via the instance references p1, p2, and p3 respectively. Everything seems fine. Reading class-wide attributes in this fashion works, but only because the attribute in question is not found in the *object scope*, so Python moves up a level to the *class scope* where it finds the count attribute and reads its value. Then, on line 19, I make a common rookie mistake. I attempt to assign a value to the class-wide count attribute via an instance reference. I'm not doing what I think I'm doing. Instead, a new instance attribute named count is created for p1. Figure 17-2 shows the results of running this program.

```

Sat Oct 5 05:46:52 EDT 2024
~/dev/cst_with_python_1st_ed/chapter17/person_project_rev1 (main)
[512:13] swodog@macos-mojave-testbed $ python3 main.py
__new__() method called...Person object created!
__init__() method called...Person object initialized!
p1 = John J Doe | Object Count: 1
__new__() method called...Person object created!
__init__() method called...Person object initialized!
p2 = Rick W Miller | Object Count: 2
__new__() method called...Person object created!
__init__() method called...Person object initialized!
p3 = Hannah J Banana | Object Count: 3
*****
p1 = John J Doe | Object Count: 100
p2 = Rick W Miller | Object Count: 3
p3 = Hannah J Banana | Object Count: 3

```

Figure 17-2: Results of Running Example 17.3

Referring to figure 17-2 — *Reading* class-wide attributes via instance references provide the naive expected behavior, however, you will run into difficulty when you attempt to *assign* a value to a class-wide attribute via an instance reference. Python allows you to create new instance attributes on-the-fly. (And new class attributes as well.) The ability to create new attributes is a powerful language feature but, as the saying goes, “With great power comes great responsibility”. This leads to the following Pro Tip: Read and write class-wide attributes via the class name.

Pro Tip: Read and write class-wide attributes via the class name

1.8 PROPERTIES VS. ATTRIBUTES

Python instance attributes are meant to be accessed directly. This may seem quite strange if you come from languages like Java, C#, or C++ that support private fields and the notion of data encapsulation. There’s no such thing as private data or data encapsulation in Python. All class and instance attributes are publicly accessible, however, there is a convention in Python, which is discussed in PEP 8, that instructs you to prefix an underscore ‘_’ to attribute names to serve as a signal to clients not to access the attribute directly because it is an implementation detail and may change in the future. Let’s add an age property to the Person class. To do this, I’ll add a date of birth (`self._dob`) instance attribute and a property named `birthday`, along with several other properties for convenience. Example 17.4 gives the revised Person class.

17.4 *person.py (rev 1)*

```

1  """Contains the definition of the Person class."""
2
3  from datetime import date
4  from datetime import datetime
5
6  class Person:
7      """Defines a Person class."""
8
9      # This is a class-wide attribute
10     # shared by all Person objects
11     # Define and initialize these first
12     count = 0
13
14     # Define the __init__() method next
15     def __init__(self, first_name:str='John',
16                 middle_name:str='J', last_name:str='Doe',
17                 date_of_birth:datetime=datetime.now()->None:
18         """Initializes Person object with known state."""
19         self.first_name = first_name
20         self.middle_name = middle_name
21         self.last_name = last_name
22         # Underscore warns clients not to access this attribute
23         self._dob = date_of_birth
24         Person.count += 1
25         if __debug__:
26             print(f'__init__() method called...Person object initialized!')
27
28     # Define properties next
29     # Group property definition with
30     # corresponding setters and deleters (if any)
31     @property
32     def birthday(self)->datetime:

```



```

33         """Return person's birthday."""
34         return self._dob
35
36     @birthday.setter
37     def birthday(self, value:datetime)->None:
38         """Set person's birthday."""
39         self._dob = value
40
41     @property
42     def full_name(self)->str:
43         """Return person's full name."""
44         return f'{self.first_name} {self.middle_name} {self.last_name}'
45
46     @property
47     def age(self)->int:
48         """Return person's age in years."""
49         today = datetime.now().date()
50         b_day = date(today.year, self._dob.month, self._dob.day)
51         adjustment = (1 if today < b_day else 0)
52         return (today.year - self._dob.year) - adjustment
53
54     @property
55     def full_name_and_age(self)->str:
56         """Return person's full name and age."""
57         return f'{self.full_name} {self.age}'
58
59     def __str__(self)->str:
60         """Returns a string representation of the object."""
61         return self.full_name_and_age
62

```

Referring to example 17.4 — I would first like to draw your attention to the overall code layout. There's a doc comment on the first line of the module followed by the `import` statements followed by the `class` definition. Within the body of the `class` definition the doc comment comes first followed by any class-wide attribute definitions, followed by the `__init__()` method which contains instance attribute definitions and any necessary initialization code. Next come property definitions, ordinary method definitions (if any) and lastly, the remaining special method definitions. This is just my personal preference.

Again from the top, on lines 3 and 4, I'm importing the `date` and `datetime` classes. To the `__init__()` method, on line 23, I have added a new instance attribute `self._dob`. On line 17, I have also added another `__init__()` method parameter named `date_of_birth`. Note that the `self._dob` instance attribute begins with a leading underscore. This signals to developers using this class not to access this attribute directly, but, like I said earlier, this is only a suggestion — a professional agreement. The message being that if you do access this attribute directly, your code may break at some point in the future if the developer of the `Person` class decides to change the name of that attribute or remove it altogether. This isn't really a problem if you are the only one working on the code, but in a team environment, or if you ever intend to release a public package on `PyPi.org`, then it becomes a serious issue.

OK, continuing on to line 31, I use the `@property` decorator to declare a read-only property named `birthday`. It returns the `self._dob` attribute. On line 36, I declare a `birthday` setter property with the `@birthday.setter` decorator. Note that the `@birthday.setter` property declares a parameter named `value`, which is assigned to the `self._dob` attribute.

Lastly, I define three more properties on lines 42, 47, and 54 respectively named `full_name`, `age`, and `full_name_and_age`. The `__str__()` special method simply returns the `self.full_name_and_age` property. OK, let's put this new version of the `Person` class through its paces. Example 17.5 lists a revised `main.py` module.

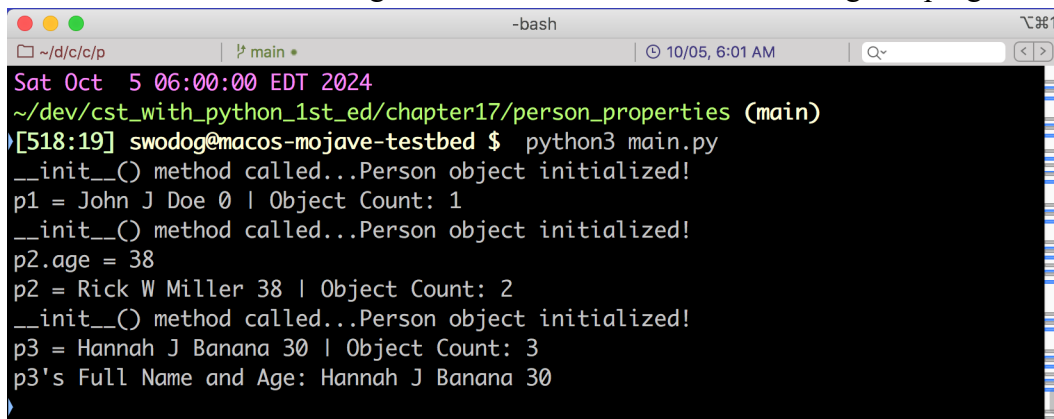
17.5 *main.py* (rev 2)

```

1  """Demonstrate object instantiation and attribute access."""
2
3  from person import Person
4  from datetime import datetime
5
6  def main():
7      p1 = Person()
8      print(f'p1 = {p1} | Object Count: {Person.count}')
9      p2 = Person('Rick', 'W', 'Miller', datetime(1986, 4, 2))
10     print(f'p2.age = {p2.age}')
11     print(f'p2 = {p2} | Object Count: {Person.count}')
12     p3 = Person()
13     p3.first_name = 'Hannah'
14     p3.middle_name = 'J'
15     p3.last_name = 'Banana'
16     p3.birthday = datetime(1994, 3, 14)
17     print(f'p3 = {p3} | Object Count: {Person.count}')
18     print(f'p3's Full Name and Age: {p3.full_name_and_age}')
19
20
21  if __name__ == '__main__':
22     main()
23

```

Referring to example 17.5 — On line 4, I'm importing the `datetime` class and use it on lines 9 and 16 to create `datetime` objects for birthdays. Something very important to notice is that although property definitions look like method definitions, they are used like ordinary attributes as shown on lines 10, 16, and 18. Figure 17-3 shows the results of running this program.



```

Sat Oct 5 06:00:00 EDT 2024
~/dev/cst_with_python_1st_ed/chapter17/person_properties (main)
[518:19] swodog@macos-mojave-testbed $ python3 main.py
__init__() method called...Person object initialized!
p1 = John J Doe 0 | Object Count: 1
__init__() method called...Person object initialized!
p2.age = 38
p2 = Rick W Miller 38 | Object Count: 2
__init__() method called...Person object initialized!
p3 = Hannah J Banana 30 | Object Count: 3
p3's Full Name and Age: Hannah J Banana 30

```

Figure 17-3: Results of Running Example 17.5

1.9 STOPPING THE DEBUG STATEMENTS

You can see from figure 17-3 that I run the `main.py` module like so:

```
python3 main.py
```

This sets the `__debug__` global variable to `True`. To set the `__debug__` global variable to `False`, add the `-O` switch (capital letter O) to the `python3` command like so:

```
python3 -O main.py
```

QUICK REVIEW

In the world of software engineering, the term *class* has several meanings depending upon the context in which it appears of which there are three we are concerned with here: object-oriented *analysis*, object-oriented *design*, and object-oriented *programming*. In the context of object-oriented *analysis & design*, a class is used to represent the concept or notion of an entity within a problem domain. In the context of object-oriented *programming*, a class refers to a programming language construct that enables a programmer to implement in code the conceptual notion of a class as discovered during analysis and design. Python provides the `class` keyword, which lets you define *user-defined types* which serve as templates for the creation of objects within your program.

A class is used to create one or more *objects* within your program. Use the `class` keyword to define a class in Python.

Instantiate an object by calling the class *constructor*, which is the name of the class followed by open-close parentheses (i.e., `Person()`).

The `__new__()` method is responsible for *object creation* and is referred to as the constructor. The `__init__()` method is responsible for *object initialization*. The `__new__()` method is optional as Python provides a default version which works fine in most situations.

Class-wide attributes are shared by all instances. Access class-wide attributes via the class name. Instance attributes are unique to each object. Access instance attributes via object references.

Python classes have no concept of encapsulation. Preceding an instance attribute with an underscore `'_'` indicates the attribute should not be accessed directly because it is an implementation detail and may change or be removed at some point in the future.

Create read-only properties with the `@property` decorator. Add a setter decorator if you need a read-write property.

Use properties if you need to perform some transformation or calculation on an instance attribute.

2 OBJECT-ORIENTED ANALYSIS, DESIGN, AND PROGRAMMING

Now that you know how to define a class and use it to create objects, I want to step back and give you an overview of object-oriented analysis, design, and programming. In this short section I will only be hitting the highlights. You would normally dive deeper into this topic in a course on Software Engineering.

In Chapter 2: An Approach To The Art Of Programming, I introduced you to the three roles you're expected to play as a software engineer: Analyst, Architect, and Programmer. Large projects may have different people performing these roles, each with highly specialized skills. In my experience, most of the work falls to the individual software engineer who must build competence and expertise in all three areas.

2.1 OBJECT-ORIENTED ANALYSIS

The analysis phase of any software development project regardless of size involves studying a system or process, either existing or planned, for the purpose of capturing functional and non-functional requirements which can later be translated into a system design.

Object-Oriented Analysis places emphasis on identifying entities that participate in the system or process being analyzed. An *entity* can be a human playing some role in the system (i.e., user, operator, data source, data receiver, etc.) or a concept within the system (i.e., customer, order, product, employee, student, etc.). Analysis may produce a one-to-one mapping between real-world objects and software entities modeled within the system.

Activities associated with object-oriented analysis include requirements gathering, object identification, inter-object relationship identification including associations between objects, compositional relationships, inheritance relationships, and object behavior identification.

Several different modeling approaches may be used to capture and enhance understanding of system requirements. These include use-case modeling, which is captured in UML Use-Case diagrams, static modeling, which is captured in UML Class diagrams, and dynamic modeling, which is captured in UML Sequence diagrams.

2.2 OBJECT-ORIENTED DESIGN

Any software design phase involves defining a system architecture comprised of detailed design models based on the requirements gathered during analysis.

Object-Oriented Design places an emphasis on adding detail to classes identified during the analysis phase including *attributes*, *methods*, *relationships*, and *interactions*. Approaches to object-oriented design include compositional design, inheritance, or a combination of both. (*In fact, as you'll learn later, there really is no object-oriented design without inheritance.*)

To speed up the design process and enhance overall system reliability, flexibility, and extensibility, experienced software engineers will employ well-known software design principles (i.e., Dependency Inversion Principle, Liskov Substitution Principle, and the Open-Closed Principle) and established software design patterns (i.e. Facade pattern, Model-View-Controller pattern, Command pattern, etc.). You will learn more about design principles and software design patterns later in the book.

The detailed design can be expressed in UML Component, Class, Sequence, and State Machine diagrams.

2.3 OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming entails using a programming language that supports the object-oriented paradigm to implement the detailed system architecture created during the design phase. Emphasis is placed on defining classes that bundle data along with the methods that process the data. Key concepts besides classes and objects include encapsulation, abstraction, inheritance, and polymorphism.

Some programming languages support OOP concepts better than others. Python, since this is a book about Python, does not support data encapsulation, which is the notion of limiting access to an object's private data. This, however, is usually not a show-stopper if the hands-off signal given with a preceding underscore is honored.

2.4 WHAT'S THE POINT?

The point of Object-Oriented Analysis, Design, and Programming (OOAD&P) is to produce a software system or application that not only works and reliably so, but is also easy to *comprehend*, *maintain*, and *evolve*.

2.4.1 EASY TO COMPREHEND

The system architecture should be easy to understand. A class must have an obvious purpose and its relationship with other classes within the system must be clear. You should be able to wrap your head around what each component within the system is doing and how it does it.

If you find yourself struggling to comprehend a system design, or if you're looking at existing code and wondering what in the name of everything that's holy is going on, then the system is poorly designed or at the very least is suboptimal.

2.4.2 EASY TO MAINTAIN

A well-designed system should be easy to maintain. The code base layout should correspond to architectural components and boundaries. By code base layout I am referring to how the project is organized. It should be obvious from the system architecture where a particular type of problem might originate. You should be able to picture in your mind where the class or classes involved with an application feature reside. (i.e, In what folders, files, and modules.)

A poorly designed system architecture leads to shortcuts. If it's a pain-in-the-ass to achieve a design objective then lazy programmers will take shortcuts. This leads to a well-documented phenomenon referred to as Code Rot which should never be allowed to gain a foothold and eradicated early and with prejudice if it does start to creep into the code base.

2.4.3 EASY TO EVOLVE

A well-designed object-oriented system should easily accommodate new features while maintaining architectural integrity. Through the use of composition, inheritance, and software design principles such as the Dependency Inversion Principle, Liskov Substitution Principle, and the Open-Closed Principle, and a small handful of software design patterns such as the Command Pattern, the Model-View-Controller Pattern, the Strategy Pattern, the Observer pattern, the Decorator pattern, and the Façade pattern, additional application features should require little if any modification to existing code. Rather, a high-level abstraction which specifies desired behavior is extended to implement the desired functionality.

2.5 HOW DO YOU ARRIVE AT A GOOD DESIGN?

Good design evolves incrementally over time. Rarely, if ever, will a developer, or a team of engineers, start with a complete, perfectly designed application architecture. Modern software development practices abhor the notion of trying to design everything up front. Of course, you must have a good idea of *what* it is you are building but the details of *how* it will be implemented can be worked out over time. An experienced software engineer will know when they have arrived at a point that requires a code reorganization to achieve their design objectives. This code reorganization activity is referred to as refactoring.

How, if you're a relatively inexperienced developer, can you tell if you need to refactor your code? I like to describe the feeling like this: *You arrive at a point where you have programmed yourself into a corner from which there is no clean escape except via a path that violates and corrupts the application architecture.* At that point you must stop, reevaluate the design, and refactor the code so that it continues to support desired design objectives. (i.e., Easy to Understand, Easy to Maintain, and Easy to Evolve.)

QUICK REVIEW

Object-Oriented Analysis places emphasis on identifying entities that participate in the system or process being analyzed. *Object-Oriented Design* places emphasis on adding detail to classes identified during the analysis phase including *attributes*, *methods*, *relationships*, and *interactions*. *Object-Oriented Programming* entails using a programming language that supports the object-oriented paradigm to implement the detailed system architecture created during the design phase.

The point of Object-Oriented Analysis, Design, and Programming (OOAD&P) is to produce a software system or application that is easy to comprehend, maintain, and evolve.

Good design evolves incrementally over time.

3 INTRODUCTION TO UNIFIED MODELING LANGUAGE [UML]

Unified Modeling Language (UML) is the de facto standard for expressing the design and implementation of object-oriented systems. The UML specification defines both a visual language which is used to specify and communicate system designs, as well as a MOF-based meta-model that specifies how software tools create, display, and share UML models. The acronym MOF stands for *Meta Object Facility*. You can learn more about UML and MOF at the Object Modeling Group's website: <https://omg.org>

3.1 UML IS EASY TO DRAW

UML diagram elements are meant to be easy to draw regardless of the tool you are using. I mostly draw UML diagrams on a white board with a dry erase marker to explain to co-workers or students what I am thinking. I have drawn UML diagrams on napkins while having coffee with colleagues, and in my Engineer's Notebook when I'm having coffee by myself. Over the years, I have drawn UML diagrams using various diagramming software like Microsoft Visio, Microsoft PowerPoint, Lucid Charts, and Draw.io. I have used full-blown UML modeling tools to automatically draw UML diagrams from an existing code base. The purpose of drawing a UML diagram is to visually communicate your design to yourself and to other humans.

3.2 A LITTLE UML GOES A LONG WAY

A complete treatment of UML is way beyond the scope of this book, however, it will be extremely helpful to use a small handful of UML diagram elements to provide you with a visual model of some of the more complex examples you will encounter throughout the remainder of this book. I will rely mostly upon *class diagrams* and *sequence diagrams*.

3.2.1 CLASS DIAGRAM

A class diagram contains one or more class elements and shows the relationship between them. A class is drawn as a rectangle which can optionally contain several inner rectangles to display class properties, attributes, and methods. Figure 17-4 shows the class diagram for the Person class defined in example 17.4.

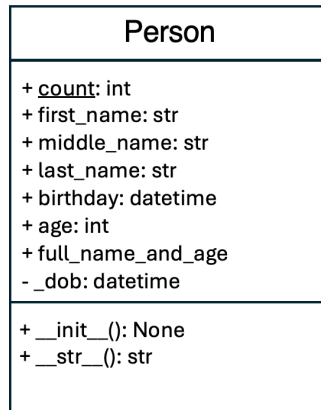


Figure 17-4: Person Class Diagram Showing Attributes and Methods

Referring to figure 17-4 — The top box contains the class name. The middle box lists attributes and properties. The plus and minus signs preceding each attribute/property (+, -) indicate public and private accessibility. As you learned earlier, all attributes, properties, and methods in a Python class are publicly accessible. Still, it's nice to be able to look at a picture of the class and see at a glance which members are considered part of an object's public interface along with those that are not. Notice that the class-wide attribute count is underlined which indicates it is a static (i.e., class-wide) variable.) The bottom box lists the methods.

3.2.1.1 ABBREVIATED CLASS DIAGRAM

I drew the Person class above using PowerPoint. Adding attributes and methods is a tedious process and in most cases these can be omitted, especially in the case where you want to focus on class relationships. The abbreviated form of the Person class diagram is given in figure 17-5.

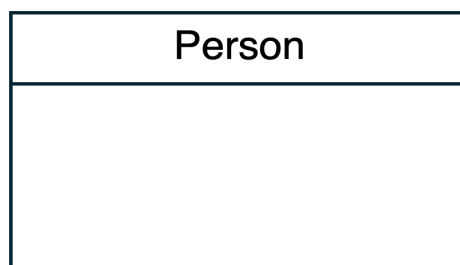


Figure 17-5: Abbreviated Person Class Diagram

Referring to figure 17-5 — I tend to use the abbreviated class form in my class diagrams simply because it's super easy to draw on a whiteboard, in a notebook, or with any computer-based drawing tool you might have.

3.2.2 SEQUENCE DIAGRAM

A sequence diagram focuses on a subset of an application architecture or process to illustrate events, method calls, and data flow between actors and entities. Example 17.5 shows a sequence diagram for the `main.py` module and the messages and events related to the variables `p1` and `p2`.

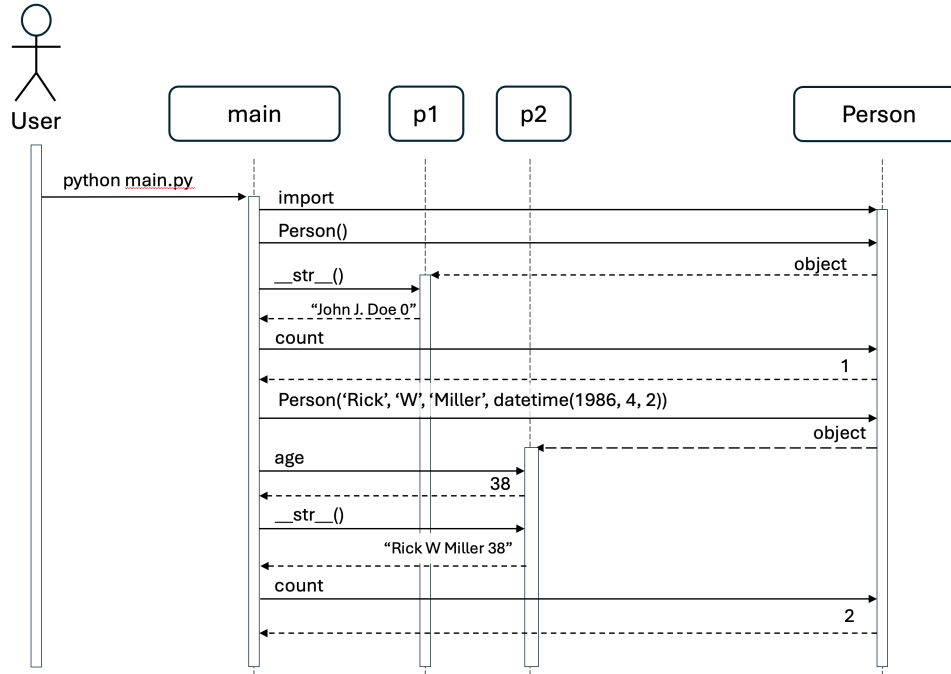


Figure 17-6: Sequence Diagram for `main.py` Module Given in Example 17.5

Referring to figure 17-6 — Actors and entities appear at the top of the sequence diagram. Their order of engagement moves from left-to-right. (*At this point you should follow along in the code while consulting the sequence diagram.*) In this diagram, the stick figure represents a UML actor, which represent a role play by a user or another system that interacts with the program. Extending down from each entity are *lifelines*. When the user runs the `main.py` program, the main module executes and imports the `Person` class from the `person` module. A call is made to the `Person()` constructor which returns an object of type `Person` and assigns the reference to the variable `p1`. With `p1` thus created, an implicit call is made to `p1.__str__()`, which returns a string representation of the `Person` object referenced by `p1`. Next, the `Person.count` attribute is accessed which results in the value `1` being returned. I leave it for you as an exercise to trace the sequence of calls related to `p2`. **Note:** I left `p3` out of this diagram in the interest of space.

3.2.2.1 THOUGHTS ON SEQUENCE DIAGRAMS

Sequence diagrams can be extremely tedious to draw manually, especially if you try to represent too much of the system in one diagram. They are, however, an extremely helpful tool for clarifying your understanding of what's happening in the code.

I rarely draw detailed sequence diagrams manually. Instead, I'll use a comprehensive UML modeling tool that can read in the source files and automatically generate the diagram. I use the same approach when I need to create comprehensive class diagrams.

QUICK REVIEW

Unified Modeling Language (UML) is used to express the design and implementation of object-oriented systems. UML defines both a set of visual diagram elements which are used to communicate the design of an object-oriented system, as well as the specification of a meta-language for use by automated UML design tools.

A little UML goes a long way. Two of the most helpful UML diagram types include class and sequence diagrams. Class diagrams show one or more classes and their relationship to each other. Sequence diagrams show system events, messages, and data flow between participating actors and objects.

4 METHODS

A *method* is a function that is defined within a class. You learned about functions in Chapter 12: Modules and Functions, so here I shall focus only on how to define and call methods. Everything you learned about functions with regards to defining parameters, passing in arguments, and returning values, applies to methods. However, before diving into the meat of this section, I want to briefly discuss the special methods and properties of the Person class listed in example 17.4.

4.1 SPECIAL METHODS

As you learned earlier, the Person class defines two *special* methods: `__init__()` and `__str__()`. The purpose of special methods is to customize object behavior with regards to how they are used with various language operators. In the case of the person class, I am, at this point at least, only interested in customizing *object initialization* and providing a *custom string representation* of a person object.

The `__init__()` method is *implicitly* called when a person instance is created as shown in the following code snippet.

```
p1 = Person()
```

The `__str__()` method is *implicitly* called when a person object is passed as an argument to the built-in `print()` function like so:

```
print(p1)
```

Note that you are technically not required to implement either of these methods, but it's considered best practice to do so. If you do not provide an implementation for the `__init__()` method, you would need to write additional code to define instance attributes.

The default implementation of the `__str__()` method returns information about an object's type and memory location. For example, removing the definition of the `__str__()` method from the Person class results in the following output when I call `print(p1)`:

```
<person.Person object at 0x106245390>
```

If you try this at home the memory location value will most certainly be different, and will be different for each person object you create, and perhaps even different every time you run the program.

Note that *special methods* are Python's way of allowing *operator overloading*, a topic I cover in greater detail in Chapter 19: Well-Behaved Objects.

4.2 PROPERTY DEFINITIONS

Referring again to the `Person` class listed in example 17.4 — The property definitions look suspiciously like method definitions but the `@property` decorator changes the semantics. By this I mean you access properties like attributes as opposed to calling them like a method.

4.3 DEFINING AND CALLING METHODS

A method is defined within the body of a class definition. A method defines at least one parameter, `self`, which is a reference to an instance. The parameter name `self` is strictly a convention. You could use any name, but I recommend sticking with `self` as all the documentation and examples on the Internet use `self`...so...just use `self`.

The biggest problem you'll face when starting to create your own classes and methods is forgetting to add the parameter `self` to your methods.

OK, to inject some fun into this conversation, I'm going to add the capability to associate `Person` objects with family relationships. See the relationship graph in figure 17-7.

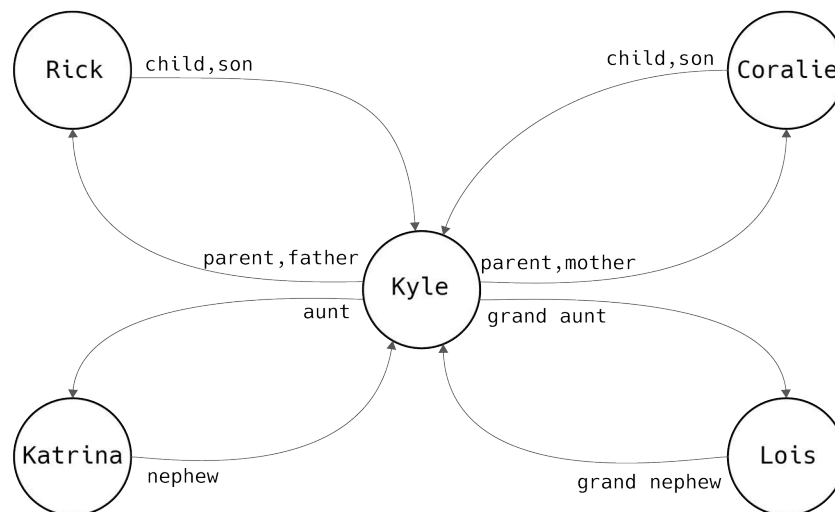


Figure 17-7: Family Relationship Graph

Referring to figure 17-7 — The graph consists of *nodes* (vertices) represented by circles and *edges*, represented by the lines connecting each node. This is a *directed graph* in that the edges have an arrow that indicates the direction of the relationship. For example, Kyle is Rick's child and Rick is Kyle's father. I have combined edges in the interest of space and orderliness.

To store a person's relationships I will add a dictionary to the `Person` class along with several methods to add and show a person's relationships. Example 17.6 gives the modified `Person` class.

17.6 *person.py*

```

1  """Contains the definition of the Person class."""
2
3  from datetime import date
4  from datetime import datetime
5
6  class Person:
7      """Defines a Person class."""
8

```

```

9     # This is a class-wide attribute
10    # shared by all Person objects
11    # Define and initialize these first
12    count = 0
13
14    # Define the __init__() method next
15    def __init__(self, first_name:str='John',
16                middle_name:str='J', last_name:str='Doe',
17                date_of_birth:datetime=datetime.now()->None:
18        """Initializes Person object with known state."""
19        self.first_name = first_name
20        self.middle_name = middle_name
21        self.last_name = last_name
22        # Underscore warns clients not to access this attribute
23        self._dob = date_of_birth
24        self._relationships = {}
25        Person.count += 1
26
27
28    # Define properties next
29    # Group property definition with
30    # corresponding setters and deleters (if any)
31    @property
32    def birthday(self)->datetime:
33        """Return person's birthday."""
34        return self._dob
35
36
37    @birthday.setter
38    def birthday(self, value:datetime)->None:
39        """Set person's birthday."""
40        self._dob = value
41
42
43    @property
44    def full_name(self)->str:
45        """Return person's full name."""
46        return f'{self.first_name} {self.middle_name} {self.last_name}'
47
48
49    @property
50    def age(self)->int:
51        """Return person's age in years."""
52        today = datetime.now().date()
53        b_day = date(today.year, self._dob.month, self._dob.day)
54        adjustment = (1 if today < b_day else 0)
55        return (today.year - self._dob.year) - adjustment
56
57
58    @property
59    def full_name_and_age(self)->str:
60        """Return person's full name and age."""
61        return f'{self.full_name} {self.age}'
62
63
64    def __str__(self)->str:
65        """Returns a string representation of the object."""
66        return self.full_name_and_age
67

```

0
0
0
1
0
0
0
1

```

68
69 def add_relationship(self, p:any, relationship:str)->None:
70     """Add relationship to p."""
71     if relationship not in self._relationships:
72         self._relationships[relationship] = []
73
74     match(relationship):
75         case 'parent' | 'father' | 'mother' \
76             | 'sibling' | 'brother' | 'sister' \
77             | 'child' | 'daughter' | 'son' \
78             | 'aunt' | 'uncle' | 'niece' | 'nephew' \
79             | 'grand aunt':
80             if isinstance(p, Person):
81                 if p not in self._relationships[relationship]:
82                     self._relationships[relationship].append(p)
83         case _:
84             self._relationships[relationship].append(p)
85
86
87 def show_relationships(self)->None:
88     """Show all relationships."""
89     for key in self._relationships:
90         for entry in self._relationships[key]:
91             print(f'{self.full_name} --> {key} : {entry}')
92

```

Referring to example 17.6 — Starting on line 24 in the body of the `__init__()` method, I have added an instance attribute named `self._relationships`, which is a dictionary that will contain a `Person` object's relationships. I have also added two new methods: `add_relationship()`, which starts on line 69, and `show_relationships()`, which starts on line 87.

The `add_relationship()` method takes two arguments: `p`, which is an object who's relationship will be added to the current person, and `relationship`, which is a string that defines the relationship. Essentially, the `add_relationship()` method works like this: The `relationship` string serves as a key in the dictionary. Objects that participate in that relationship are stored in a list. On line 71, I check to see if the `relationship` key exists in the dictionary, and if not, I create an empty list using that key. The `match` statement starting on line 74 checks the `relationship` string. If it matches one of the family relationships listed in the first case I then check to ensure it's a `Person` object using the `isinstance()` function. If it is, I then check to see if it is already in the relationship list, and if not, I append it to the list. The default case simply adds `p` to the indicated relationship with no questions asked.

The `show_relationships()` method begins on line 87. It simply iterates over the `self._relationships` dictionary and prints the relationships to the console.

Example 17.7 shows these methods in action.

17.7 main.py

```

1     """Demonstrate method calling."""
2
3     from person import Person
4     from datetime import datetime
5
6     def main():
7         # Create Person Objects
8         kyle = Person('Kyle', 'V', 'Miller', datetime(2016, 8, 12))
9         rick = Person('Rick', 'W', 'Miller', datetime(1986, 4, 2))
10        coralie = Person('Coralie', 'S', 'Miller', datetime(1989, 10, 6))
11        katrina = Person('Katrina', 'M', 'Powell', datetime(1990, 12, 3))

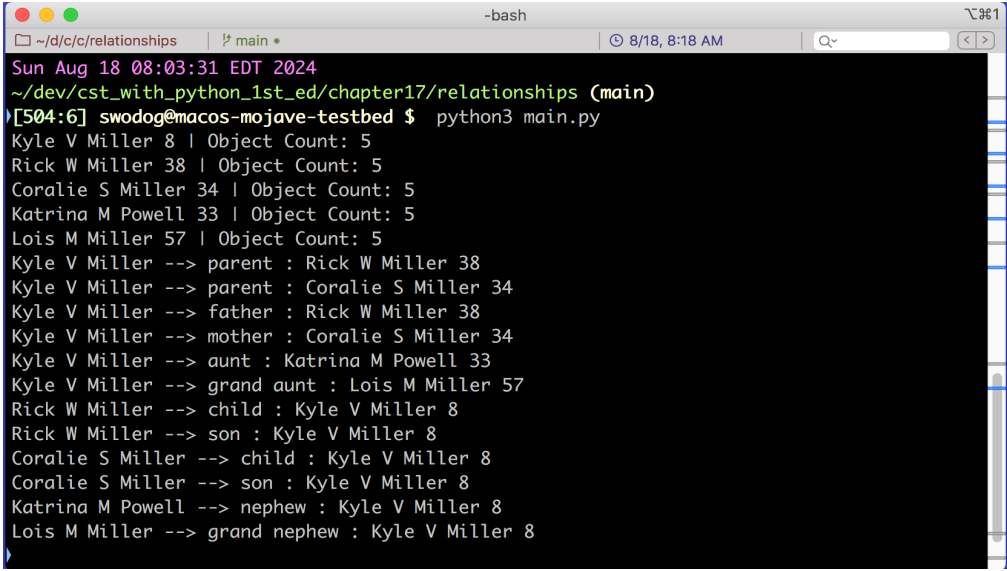
```

```

12     lois = Person('Lois', 'M', 'Miller', datetime(1966, 10, 20))
13     # Print Person Info
14     print(f'{kyle} | Object Count: {Person.count}')
15     print(f'{rick} | Object Count: {Person.count}')
16     print(f'{coralie} | Object Count: {Person.count}')
17     print(f'{katrina} | Object Count: {Person.count}')
18     print(f'{lois} | Object Count: {Person.count}')
19     # Add Relationships
20     kyle.add_relationship(rick, 'parent')
21     kyle.add_relationship(rick, 'father')
22     rick.add_relationship(kyle, 'child')
23     rick.add_relationship(kyle, 'son')
24     kyle.add_relationship(coralie, 'parent')
25     kyle.add_relationship(coralie, 'mother')
26     coralie.add_relationship(kyle, 'child')
27     coralie.add_relationship(kyle, 'son')
28     kyle.add_relationship(katrina, 'aunt')
29     katrina.add_relationship(kyle, 'nephew')
30     kyle.add_relationship(lois, 'grand aunt')
31     lois.add_relationship(kyle, 'grand nephew')
32
33     # Show Relationships
34     kyle.show_relationships()
35     rick.show_relationships()
36     coralie.show_relationships()
37     katrina.show_relationships()
38     lois.show_relationships()
39
40
41     if __name__ == '__main__':
42         main()
43

```

Referring to example 17.7 — The first thing I do on lines 8 through 12 is create five Person objects: kyle, rick, coralie, katrina, and lois. On lines 14 through 18, I print the each Person object’s information. Next, I add the relationships as shown in figure 17-7, then print the relationships for each Person object to the console. Figure 17-8 shows the results of running this program.



```

Sun Aug 18 08:03:31 EDT 2024
~/dev/cst_with_python_1st_ed/chapter17/relationships (main)
[504:6] swodog@macos-mojave-testbed $ python3 main.py
Kyle V Miller 8 | Object Count: 5
Rick W Miller 38 | Object Count: 5
Coralie S Miller 34 | Object Count: 5
Katrina M Powell 33 | Object Count: 5
Lois M Miller 57 | Object Count: 5
Kyle V Miller --> parent : Rick W Miller 38
Kyle V Miller --> parent : Coralie S Miller 34
Kyle V Miller --> father : Rick W Miller 38
Kyle V Miller --> mother : Coralie S Miller 34
Kyle V Miller --> aunt : Katrina M Powell 33
Kyle V Miller --> grand aunt : Lois M Miller 57
Rick W Miller --> child : Kyle V Miller 8
Rick W Miller --> son : Kyle V Miller 8
Coralie S Miller --> child : Kyle V Miller 8
Coralie S Miller --> son : Kyle V Miller 8
Katrina M Powell --> nephew : Kyle V Miller 8
Lois M Miller --> grand nephew : Kyle V Miller 8

```

Figure 17-8: Results of Running Example 17.7

Again referring to example 17.7 — Note that a method is called on a particular object using the dot `.` operator and the open and close parentheses `()`. Arguments passed to the method appear between the parentheses (i.e., `kyle.add_relationship(rick, 'parent')`). Within the method definition, the name `self` represents that instance. The terms *object* and *instance* are synonymous. Every *instance of Person*, or to say it another way, every *Person object*, contains its own copy of its instance attributes. When you call a method via a reference to an object (i.e., `kyle`) the method (i.e., `add_reference(rick, 'parent')`) works on that object's data.

I like to project myself into the code when I am programming. When you are defining a class, you can think of yourself as being "*in the object*" and the name `self` represents the *object* you are in and thus any instance attributes you define belong to that object's data.

QUICK REVIEW

Methods are functions that are defined within a class. All methods have at least one parameter named `self`, which represents an instance of the class. Everything you learned about functions regarding parameter definition, argument passing, and returning values, applies to methods as well.

Use the dot `.` operator and the open and closed parentheses `()` to call a method via an object reference.

SUMMARY

In the world of software engineering, the term *class* has several meanings depending upon the context in which it appears of which there are three we are concerned with here: object-oriented *analysis*, object-oriented *design*, and object-oriented *programming*. In the context of object-oriented *analysis & design*, a class is used to represent the concept or notion of an entity within a problem domain. In the context of object-oriented *programming*, a class refers to a programming language construct that enables a programmer to implement in code the conceptual notion of a class as discovered during analysis and design. Python provides the `class` keyword, which lets you define *user-defined types* which serve as templates for the creation of objects within your program.

A class is used to create one or more *objects* within your program. Use the `class` keyword to define a class in Python.

Instantiate an object by calling the class *constructor*, which is the name of the class followed by open-close parentheses (i.e., `Person()`).

The `__new__()` method is responsible for *object creation* and is referred to as the constructor. The `__init__()` method is responsible for *object initialization*. The `__new__()` method is optional as Python provides a default version which works fine in most situations.

Class-wide attributes are shared by all instances. Access class-wide attributes via the class name. Instance attributes are unique to each object. Access instance attributes via object references.

Python classes have no concept of encapsulation. Preceding an instance attribute with an underscore `'_'` indicates the attribute should not be accessed directly because it is an implementation detail and may change or be removed at some point in the future.

Create read-only properties with the `@property` decorator. Add a setter decorator if you need a read-write property.

Use properties if you need to perform some transformation or calculation on an instance attribute.

Object-Oriented Analysis places emphasis on identifying entities that participate in the system or process being analyzed. *Object-Oriented Design* places emphasis on adding detail to classes identified during the analysis phase including *attributes*, *methods*, *relationships*, and *interactions*. *Object-Oriented Programming* entails using a programming language that supports the object-oriented paradigm to implement the detailed system architecture created during the design phase.

The point of Object-Oriented Analysis, Design, and Programming (OOAD&P) is to produce a software system or application that is easy to comprehend, maintain, and evolve.

Good design evolves incrementally over time.

Unified Modeling Language (UML) is used to express the design and implementation of object-oriented systems. UML defines both a set of visual diagram elements which are used to communicate the design of an object-oriented system, as well as the specification of a meta-language for use by automated UML design tools.

A little UML goes a long way. Two of the most helpful UML diagram types include class and sequence diagrams. Class diagrams show one or more classes and their relationship to each other. Sequence diagrams show system events, messages, and data flow between participating actors and objects.

Methods are functions that are defined within a class. All methods have at least one parameter named `self`, which represents an instance of the class. Everything you learned about functions regarding parameter definition, argument passing, and returning values, applies to methods as well.

Use the dot `.` operator and the open and closed parentheses `()` to call a method via an object reference.

SKILL-BUILDING EXERCISES

- 1. SOLID Design Principles:** Coined by Robert C. Martin, SOLID is a mnemonic for five software design principles. Research the five design principles represented by SOLID. Write a brief description of each and state how each principle contributes to understandable, flexible, and maintainable code.
- 2. History of Object-Oriented Programming:** Research the history of object-oriented programming with an eye towards answering the following questions: What were the motivating factors in the development of object-oriented analysis, design, and programming techniques? What was the first object-oriented programming language? Who were the key players in the development of object-oriented programming languages and techniques and what role did they play?
- 3. Unified Modeling Language:** Dive deeper into the purpose and use of the Unified Modeling Language. You can start by visiting the UML website: <https://www.uml.org>. List the different types of UML diagrams and their purpose. What's the relationship between UML and SysML?

4. **Python Support For Object-Oriented Programming:** How does Python compare to other programming languages that support object-oriented programming?
5. **Python Data Model:** Dive deeper into the Python data model: <https://docs.python.org/3/reference/datamodel.html>.
6. **Properties Vs. Attributes:** Dive deeper into properties defined with the `@property` decorator and instance attributes. Write some code to test what happens when you define a property with the same name as an instance attribute?
7. **Data Classes:** Research Python data classes. What special feature do data classes support? When do you think it would be appropriate to use data classes in a program?
8. **Python Special Methods.** The `__new()`, `__init__()`, and `__str__()` defined in the `Person` class are examples of Python special methods. Create a list of all Python special methods and briefly describe each method's purpose.
9. **UML Modeling Tools: Research UML modeling tools.** Find tools that let's you generate class and sequence diagrams from an existing code base. Write a brief paper summarizing your experience and impressions of each tool.
10. **The Python `__dict__` Special Property:** Research the `__dict__` property. Explain in your own words it's purpose and use.

SUGGESTED PROJECTS

1. **Home Inventory Program:** Write a program that lets you keep track of the items in your home. Your program should allow users to create multiple inventories, add items to an inventory, delete items from an inventory, list the different inventories, select an inventory, and list the items contained within the selected inventory. Provide a console-based menu with which to interact with the application. Save inventory data to a file in JSON format. Your final program should consist of at least three source files, one of which must be `main.py`, which will be how you run the program, one class must be focused on providing the user interface, and the third implements the primary application features.
2. **Library Management System:** Write a program that allows users to manage library assets. First, conduct an initial analysis and design to identify entities that will interact with the system, the types of operations they need to perform, and what types of assets will be included in the collection. Implement a proof-of-concept that tracks books by their location. Use the Library of Congress numbering system. Allow users to enter books into the system and list books and their location.
3. **Sales Tracker:** Write a program that would allow a manager to track an employee's sales. The program should allow the manager to add employees and associate a sale, including items and

amount, to an employee. The program must be able to read and write sales data to disk in JSON format.

- 4. Engine Simulation:** Write a basic engine simulation program that models an engine from several different parts including a water pump, fuel pump, oxygen sensor, and temperature sensor. Each component part has an attribute named `enabled` that can be set to `true` or `false`. An engine will have a `check_engine()`, `start()`, and `stop()` methods. The `check_engine()` method should check the status of each engine part and return `true` if all parts are functioning properly, otherwise it should return `false`. The engine `start()` method must call the `check_engine()` method and if all attached parts are working properly, the engine state can be changed to `running`.
- 5. Contact List:** Write a simple program that allows users to add contacts to a list and save the contact list to file in JSON format. Provide a console-based menu-driven user interface that lets users add, list, search for, display details, and delete contacts. Use the `Person` class presented in this chapter as a starting point and modify it as necessary to represent a contact with phone number, email, and home address.

SELF-TEST QUESTIONS

1. What keyword is used to define a class?
2. What's the difference between instance attributes and class attributes?
3. How should class attributes be accessed? What happens if you assign a value to a class attribute via an instance variable?
4. What's the difference between a function and a method?
5. What's the purpose of the `self` parameter?
6. What's the purpose of the `__new__()` and `__init__()` special methods?
7. How are instance attributes meant to be accessed?
8. You are reading some source code you found in an interesting project on GitHub and you see an instance attribute name that begins with a leading underscore. What does this mean?
9. What's the purpose of the `@property` decorator? What's the difference between properties and attributes?
10. How do you define a property setter?

REFERENCES

The Python Data Model, <https://docs.python.org/3/reference/datamodel.html>

Python Classes, <https://docs.python.org/3/tutorial/classes.html>

PEP 8 — Style Guide for Python Code, <https://peps.python.org/pep-0008/>

UML Guide, Object Management Group, <https://www.omg.org>

UML Specification 2.5.1, <https://www.omg.org/spec/UML/2.5.1/About-UML>

Extreme Programming, <http://www.extremeprogramming.org>

Enough Rope To Shoot Yourself In The Foot: Rules for C and C++ Programming, Allen I. Holub, McGraw-Hill, New York, ISBN: 0-07-029689-8, Via Internet Archive: <https://archive.org/details/enoughropetoshoo0000holu>

NOTES
