Learning Objectives

- **Define polymorphism**
- **Explain how method overriding is an example of polymorphism**
- **Overload a method**
- **Overload an operator**
- **Define duck typing**

Method Overriding

What is Polymorphism?

Polymorphism is a concept in object-oriented programming in which a single interface is applicable with different types. Often this means similar operations are grouped together with the same name. However, these operations with the same name will produce different results. You have already encountered a few examples of polymorphism. Enter the following code into the IDE.

```
a = 5
b = 10
print(a + b)

c = '5'
d = '10'
print(c + d)
```

Notice how the plus operator can add together two numbers and concatenate two strings. You have a single interface (the plus operator) working with different data types (integers and strings). This is an example of polymorphism.

challenge

Try this variation:

Change your code to look like this:

```
a = 5
b = '10'
print(a + b)
```

▼ **Why is there an error?**
Polymorphism allows Python to use the plus operator with different data types, but that does not mean that the plus operator can be used with all data types. The example above causes an error message because the plus operator cannot be used with an integer and a string. There are limits to polymorphism.

Method Overriding

Method overriding is another example of polymorphism that you have already seen. Overriding a method means that you have two methods with the same name, but they perform different tasks. Again you see a single interface (the method name) being used with different forms (the parent class and the child class).

```
class Alpha:
  def show(self):
    print("I am from class Alpha")

class Bravo(Alpha):
  def show(self):
    print("I am from class Bravo")

test_object = Alpha()
test_object.show()
```

As expected, the script prints `I am from class Alpha`. Now change the line of code in which you instantiate the object `test_object`. Make no other changes and run the code again.

```
test_object = Bravo()
```

Now the script prints `I am from class Bravo`. The method call did not change, but the output did. A single interface (the `.show()` method) works with multiple data types (`Alpha` and `Bravo`). This is why method overriding is an example of polymorphism.

challenge

Try this variation:

- Create and overload the method `hello` that prints `Hello from Alpha` and `Hello from Bravo`. Test the method with both class types.
  ▼ **Solution**

```
class Alpha:
  def show(self):
    print("I am from class Alpha")

  def hello(self):
    print("Hello from Alpha")

class Bravo(Alpha):
  def show(self):
    print("I am from class Bravo")

  def hello(self):
    print("Hello from Bravo")

test_object = Alpha()
test_object.hello()
```

Method Overloading

Method Overloading

Method overloading is another example of polymorphism. Method overloading occurs when you have a single method name that can take different sets of parameters. Imagine you want to write the method `sum` that can sum up to three numbers. The math involved with three parameters is slightly different that two parameters, which is different from 1 parameter, etc. Traditionally, if you declare a method that takes three parameters but only pass two, Python will throw an error message. Instead let's create a class that has two `sum` methods; one with two parameters and another with three parameters.

```
class TestClass:
  def sum(self, a, b, c):
    return a + b + c

  def sum(self, a, b):
    return a + b

obj = TestClass()
print(obj.sum(1, 2))
```

So far, everything looks good. Change the `print` statement such that you are passing `1`, `2`, and `3` as parameters for the `sum` method.

```
print(obj.sum(1, 2, 3))
```

You should see the error message that the `sum` method takes three parameters, but was passed four. What about the first method named `sum`? It has four parameters: `self`, `a`, `b`, and `c`. Why did Python not determine that this method should be used instead? When two or more methods have the same name, Python only recognizes the last method. All of the others are ignored. That is why you got an error message. Python only recognizes the second `sum` method.

Instead, you should create a method with optional parameters. Set the default for each parameter to `None`. Test to see if each parameter is not `None`. If this is true, the method was passed three parameters, and you return the sum of the three parameters. If the first two parameters are not `None` then

return the sum of the two parameters. If only one parameter is not `None` then return the parameter. Finally, if all of the parameters are `None` then return 0. By structuring the parameters in this way, you can overload the `sum` method to work in many different situations. This is how method overloading promotes polymorphism.

```
class TestClass:
  def sum(self, a = None, b = None, c = None):
    if a is not None and b is not None and c is not None:
      return a + b + c
    elif a is not None and b is not None:
      return a + b
    elif a is not None:
      return a
    else:
      return 0

obj = TestClass()
print(obj.sum())
print(obj.sum(1))
print(obj.sum(1, 2))
print(obj.sum(1, 2, 3))
```

▼ Method overloading in other languages

Some programming languages, like Java, allow you to have multiple methods with the same name and different type signatures. The compiler will figure out which method definition to use based on the parameters in the method call. This type of method overloading is much more common than the way Python implements method overloading. **Note:** do not enter this code into the IDE; it will not run.

```
class Main {
  static int sumMethod(int a, int b) {
    return a + b;
  }

  static int sumMethod(int a, int b, int c) {
    return a + b + c;
  }

  public static void main(String[] args) {
    System.out.println(sumMethod(1, 2));
    System.out.println(sumMethod(1, 2, 3));
  }
}
```

challenge

Try this variation:

- Add on to the `sum` method such that it can take up to five numbers as parameters. Be sure to test all possible method calls.

  ▼ **Solution**

  The compound conditionals have been broken up over several lines to help with legibility. To do this, you need to use the parentheses.

```
class TestClass:
  def sum(self, a = None, b = None, c = None, d = None, e =
        None):
    if (a is not None and
        b is not None and
        c is not None and
        d is not None and
        e is not None):
      return a + b + c + d + e
    elif (a is not None and
          b is not None and
          c is not None and
          d is not None):
      return a + b + c + d
    elif (a is not None and
          b is not None and
          c is not None):
      return a + b + c
    elif a is not None and b is not None:
      return a + b
    elif a is not None:
      return a
    else:
      return 0

obj = TestClass()
print(obj.sum())
print(obj.sum(1))
print(obj.sum(1, 2))
print(obj.sum(1, 2, 3))
print(obj.sum(1, 2, 3, 4))
print(obj.sum(1, 2, 3, 4, 5))
```

The None Type

Why not replace `None` as the default parameter with `0`? After all, zero represents the lack of a value. Enter the following code in the IDE and run the script.

```
class TestClass:
  def sum(self, a = 0, b = 0, c = 0):
    if a != 0 and b != 0 and c != 0:
      return a + b + c
    elif a != 0 and b != 0:
      return a + b
    elif a != 0:
      return a
    else:
      return 0

obj = TestClass()
print(obj.sum(0, 2))
```

The answer should be `2`. Instead, Python returned `0`. This is because zero is not nothing. Zero is an integer. When you want to have a placeholder for the absence of a value, `None` is the preferable choice. This is a common practice in Python.

Operator Overloading

Operator Overloading

You saw in the beginning of this chapter that the plus operator can be used with strings and integers. When a single operator can be used with many data types, this is called operator overloading. Operator overloading is another example of polymorphism.

The plus operator is not the only built-in operator that is overloaded. The multiplication operator can also work with integers and strings. **Note**, you cannot use the multiplication operator with two strings.

```
my_string = "polymorphism"
num1 = 3
num2 = 5
print(num1 * num2)
print(my_string * num1)
```

In Python, everything is an object. So when you write `num1 * num2` what you are really saying is that the integer class (`int`) is calling the multiply method (`__mul__`), and `num1` and `num2` are the parameters for this method. Here is what that looks like in code:

```
num1 = 3
num2 = 5
print(int.__mul__(num1, num2))
```

challenge

Try this variation:

- Change the `print` statement to `print(int.__add__(num1, num2))`
- Change the `print` statement to `print(int.__sub__(num1, num2))`
- Change the `print` statement to `print(int.__truediv__(num1, num2))`

Operator Overloading and User-Defined Classes

You have seen how you can overload operators for built-in classes, but you can also overload operators for classes that you create. Imagine there is a class called `FinancialAccount` which has two subclasses `BankAccount` and `InvestmentAccount`. Create instances of each subclass.

```
class FinancialAccount:
  def __init__(self, amount):
    self.account = amount

class BankAccount(FinancialAccount):
  pass

class InvestmentAccount(FinancialAccount):
  pass

my_banking = BankAccount(500)
my_investments = InvestmentAccount(750)
```

What we would like to be able to do is say `my_banking + my_investments` and have Python calculate the sum of the `account` attributes. We want the plus operator to be overloaded for both `BankAccount` and `InvestmentAccount`. Overload the operator in the parent class so that both subclasses inherit this behavior. Then print the result.

```
class FinancialAccount:
  def __init__(self, amount):
    self.account = amount

  def __add__(self, other):
    return self.account + other.account

class BankAccount(FinancialAccount):
  pass

class InvestmentAccount(FinancialAccount):
  pass

my_banking = BankAccount(500)
my_investments = InvestmentAccount(750)
print(my_investments + my_banking)
```

challenge

Try this variation:

- Remove the overloaded operator from the `FinancialAccount` class, and overload the plus operator in the `BankAccount` class.

```
class FinancialAccount:
  def __init__(self, amount):
    self.account = amount

class BankAccount(FinancialAccount):
  def __add__(self, other):
    return self.account + other.account
```

▼ **Why is there an error message?**

When you overload the plus operator in the `BankAccount` class, the plus operator will now work on all instances of the `BankAccount` class. `my_investments` is **not** an instance of the `BankAccount` class; it is an instance of the `InvestmentAccount` class. The objects `my_banking` and `my_investing` might have different types, but they are both instances of the `FinancialAccount` class. Overloaded operators in a parent class work with the child classes.

Magic Methods

Magic Methods

You may have noticed that all of the methods for operator overloading had leading and trailing double underscores. These methods are called "dunder" methods for short (`__add__` would be pronounced "dunder add"). These methods are also called magic methods. Here is a list of some common operators and their associated magic method name. You can learn about all of the magic methods here.

Operator	Method Name
+	`object.__add__(self, other)`
-	`object.__sub__(self, other)`
*	`object.__mul__(self, other)`
/	`object.__truediv__(self, other)`
//	`object.__floordiv__(self, other)`
<	`object.__lt__(self, other)`
<=	`object.__le__(self, other)`
==	`object.__eq__(self, other)`
!=	`object.__ne__(self, other)`
>	`object.__gt__(self, other)`
>=	`object.__ge__(self, other)`

Use the classes from the previous page and overload the equality operator to check if the the two accounts are equal. The magic method for equality is `__eq__`.

```
class FinancialAccount:
  def __init__(self, amount):
    self.account = amount

  def __eq__(self, other):
    return self.account == other.account

class BankAccount(FinancialAccount):
  pass

class InvestmentAccount(FinancialAccount):
  pass

my_banking = BankAccount(500)
my_investments = InvestmentAccount(750)
print(my_investments == my_banking)
```

challenge

Try this variation:

- Overload the division operator so you can divide `my_banking` by `my_investments` (or vice versa).

  ▼ **Solution**

```
class FinancialAccount:
  def __init__(self, amount):
    self.account = amount

  def __truediv__(self, other):
    return self.account / other.account

class BankAccount(FinancialAccount):
  pass

class InvestmentAccount(FinancialAccount):
  pass

my_banking = BankAccount(500)
my_investments = InvestmentAccount(750)
print(my_investments / my_banking)
```

- Overload the floor division operator (`//`) to work with `my_banking` and `my_investments`. **Remember** floor division does division and then truncates the answer to return an whole number value.

  ▼ **Solution**

```
class FinancialAccount:
  def __init__(self, amount):
    self.account = amount

  def __floordiv__(self, other):
    return self.account // other.account

class BankAccount(FinancialAccount):
  pass

class InvestmentAccount(FinancialAccount):
  pass

my_banking = BankAccount(500)
my_investments = InvestmentAccount(750)
print(my_investments // my_banking)
```

Duck Typing

Duck Typing

Duck typing is used to determine the suitability of an object not based on what it is, but based on what it does. Look at the following code. There are two totally unrelated classes, a baseball player and a song. An instance from each class is passed to the function `print_hit` which prints the result of the `hit` method.

```
import random

class BaseballPlayer:
  def hit(self):
    """Generate a random integer 1 to 4 and return the type of
       hit"""
    total_bases = random.randint(1, 4)
    if total_bases == 1:
      return "single"
    elif total_bases == 2:
      return "double"
    elif total_bases == 3:
      return "triple"
    else:
      return "home run"

class Song:
  def __init__(self, title, ranking):
    self.title = title
    self.ranking = ranking

  def hit(self):
    """A song is a hit if it appeared in a top 40 chart"""
    if self.ranking <= 40:
      return f"{self.title} is a hit song"
    else:
      return f"{self.title} is not a hit song"

def print_hit(obj):
  print(obj.hit())

my_player = BaseballPlayer()
my_song = Song("Hey Jude", 12)

print_hit(my_player)
print_hit(my_song)
```

Duck typing gets its name from the expression, “If it walks like a duck and talks like a duck, then it must be a duck.” We do not care if it really is a duck as long as it acts like a duck. The `print_hit` function does not care if the object is has the type `BaseballPlayer` or `Song`. It only cares if the object has a `hit` method. Duck typing is an example of polymorphism because, in this case, a single function works with objects of different types.

challenge

Try this variation:

- Add the class `Boxer` to the class above. This class should have a `hit` method that returns the string `"jab"`. Pass an object of type `Boxer` to the `print_hit` function.

  ▼ **Solution**

  Your code should look something like this.

```
class Boxer:
  def hit(self):
    return "jab"

my_boxer = Boxer()
print_hit(my_boxer)
```

Handling Errors

Since `print_hit` only cares about the `hit` method, it is possible to send an object to the function that does not have a `hit` method. In this case, the program would crash.

```
import random

class BaseballPlayer:
  def hit(self):
    """Generate a random integer 1 to 4 and return the type of
       hit"""
    total_bases = random.randint(1, 4)
    if total_bases == 1:
      return "single"
    elif total_bases == 2:
      return "double"
    elif total_bases == 3:
      return "triple"
    else:
      return "home run"

class Dancer:
  def pirouette(self):
    return "Spin, spin, spin"

def print_hit(obj):
  print(obj.hit())

my_player = BaseballPlayer()
my_dancer = Dancer()

print_hit(my_player)
print_hit(my_dancer)
```

You should see an `AttributeError: 'Dancer' object has no attribute 'hit'`. A common response is to check and make sure the object has the proper type before executing the `hit` method. That would violate the spirit of duck typing. Instead, call the `hit` method, and if there is a problem then respond to the error. Modify the `print_hit` function to use a `try except` block as shown below.

```
def print_hit(obj):
  try:
    print(obj.hit())
  except AttributeError as e:
    print(e)
```

Using `try except` allows you to determine the suitability of an object based on its functionality (`hit` method), but also allows you to handle an error without crashing the program.

challenge

Try this variation:

- Assume the following code

```
class Bird:
  def fly(self):
    return "I am flapping my wings"

class Car:
  def drive(self):
    return "My wheels are turning"
```

- Create the method `print_fly` that uses duck typing to print the `fly` method. The function should also handle an `AttributeError`. Test the function with objects from the `Bird` and `Car` classes.

  ▼ **Solution**

```
def print_fly(obj):
  try:
    print(obj.fly())
  except AttributeError as e:
    print(e)

my_bird = Bird()
my_car = Car()
print_fly(my_bird)
print_fly(my_car)
```

Polymorphism Formative Assessment 1

Polymorphism Formative Assessment 2