Learning Objectives - Methods

- **Define the term instance method**

- **Convert a function that modifies an object into an instance method**

- **Explain how `self` is used in methods**

- **Demonstrate the syntax for calling a method**

Methods vs Functions

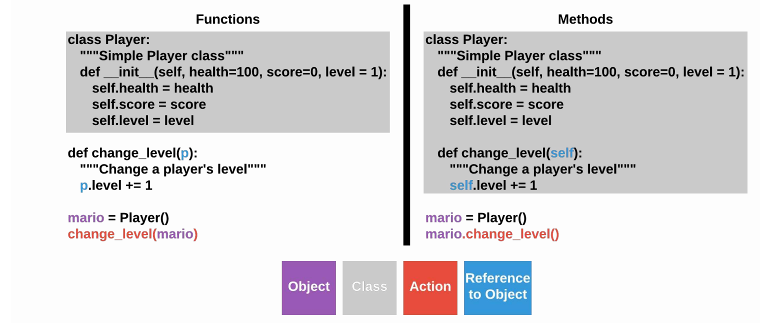
Methods

Back in the introduction to classes and objects lesson, a class was defined as "a collection of data and the actions that can modify the data." The constructor built the "collection of data", but nothing in the class modified the data. Instead, external functions were used to modify the object.

Instead of functions, methods should be used to modify an object. Think of a method as an function that is attached to an object. The instance method is the most common type of method. Notice how instance methods are declared inside of the class. The keyword `self` is used to represent the instance being modified by the method. These methods are called instance methods because they have access to the instance variables (the attributes declared in the constructor). Methods are invoked using dot-notation.

▼ **Method versus Instance Method**

Instance methods are the most common type of methods in Python. They are so common that when programmers say "method" they often mean an instance method.



Functions

```
class Player:
   """Simple Player class"""
   def __init__(self, health=100, score=0, level = 1):
      self.health = health
      self.score = score
      self.level = level

def change_level(p):
   """Change a player's level"""
   p.level += 1

mario = Player()
change_level(mario)
```

Methods

```
class Player:
   """Simple Player class"""
   def __init__(self, health=100, score=0, level = 1):
      self.health = health
      self.score = score
      self.level = level

   def change_level(self):
      """Change a player's level"""
      self.level += 1

mario = Player()
mario.change_level()
```

Functions vs Methods

When mutability was first introduced, you made a `Player` class with a few functions. You are now going to transform these functions into methods. The `Player` class will be defined just as before. This time, however, `print_player` will be a part of class (indented to match the constructor), `self` replaces `p` to represent the `Player` object being modified, and the method is called using dot-notation.

```
class Player:
  """Simple player class"""
  def __init__(self, health=100, score=0, level=1):
    self.health = health
    self.score = score
    self.level = level

  def print_player(self):
    """Print the status of a player"""
    if self.health <= 0:
      print(f"This player is dead. They died on level
        {self.level} with a score of {self.score}.")
    else:
      print(f"This player has {self.health} health, a score of
        {self.score}, and is on level {self.level}.")

mario = Player()
mario.print_player()
```

▼ **What does `self` mean?**

`self` is required as the first parameter for instance methods. It gives the method access to the instance variables. Without `self`, `print_player` would not be able to “see” the `health`, `score`, and `level` instance variables for `mario`. Remove `self` from the `print_player` definition, and Python will return an error message.

challenge

Try this variation:

Call `print_player` like this?

```
mario = Player()
print_player(mario)
```

▼ **Why did this generate an error?**

Python says that `print_player` is not defined even though the definition is on line 8. Because nothing comes before `print_player`, Python assumes that this is a function. However, `print_player` is indented inside the `Player` class, which means it is a method. Methods must be called with dot-notation like `mario.print_player()`.

More Player Methods

The next methods to add to the `Player` class are those to print the `health` and `level` of the `Player` instance. Start with the method `change_health`. This method takes `self` and the `amount` of change as parameters. `change_health` will add `amount` to the `health` attribute. If a player’s health increases, `amount` is positive. If their health decreases, `amount` is negative. Remember, make sure `change_health` is indented so that it is a part of the `Player` class.

```
  def change_health(self, amount):
    """Change a player's health"""
    self.health += amount
```

The method `change_level` is going to be similar to `change_health` except for one difference. `change_level` only needs one parameter, `self`. In video games, players go up in levels; rarely do they decrease. So the `level` attribute will increase by one when the method is called. This method also needs to be indented so it is a part of the `Player` class.

```
  def change_level(self):
    """Change a player's level"""
    self.level += 1
```

challenge

Try these variations:

- Call change_health and chagne_level for mario, and then print the player to make sure the methods work.

  ▼ **One possible solution**

  ```
  mario = Player()
  mario.print_player()
  mario.change_health(-10)
  mario.change_level()
  mario.print_player()
  ```

- Create a method to change a player's score?

  ▼ **One possible solution**

  ```
  def change_score(self, amount):
    """Change a player's score"""
    self.score += amount
  ```

▼ **Why learn about functions that modify objects when Python has methods?**

It might seem like a waste of time to learn how to write functions that modify objects. But this approach builds upon concepts you have already seen — functions and objects. This allows you to understand mutability without having to worry about methods. Once you understand how these ideas work, transforming a function into a method is much simpler. Functions that modify objects serve as an intermediary step on the way to learning about methods.

More Methods

More on Methods and Objects

Changes to objects should happen exclusively through methods. This makes your code easier to organize and easier for others to understand. Imagine you are going to create a class that keeps track of a meal. In this case, a meal can be thought of as all of the drinks, appetizers, courses, and desserts served. Each one of these categories will become an instance variable. Assign each attribute an empty list with the constructor, and create an instance of the `Meal` class.

```
class Meal:
  """Class to represent a meal"""
  def __init__(self):
    self.drinks = []
    self.appetizers = []
    self.main_course = []
    self.desserts = []

dinner = Meal()
```

Next, add a method to add a drink to `dinner`. `self.drinks` is used to access the list of drinks, and `.append` is the command to add an element to the list. So `self.drinks.append(d)` adds the drink `d` to the list `drinks`. Test your code to make sure it is working as expected. Remember to indent `add_drink` so that it is a part of the `Meal` class.

```
  def add_drink(self, d):
    """Add a drink (d) to the meal (self)"""
    self.drinks.append(d)

dinner = Meal()
dinner.add_drink("water")
print(dinner.drinks)
```

Now create the `add_appetizer` method to the class. Like the method above, `add_appetizer` accepts a string as a parameter and appends it to the `appetizers` attribute. Add `"bruschetta"` to the `dinner` instance and print it.

```
  def add_appetizer(self, a):
    """Add an appetizer (a) to the meal (self)"""
    self.appetizers.append(a)

dinner = Meal()
dinner.add_drink("water")
print(dinner.drinks)
dinner.add_appetizer("bruschetta")
print(dinner.appetizers)
```

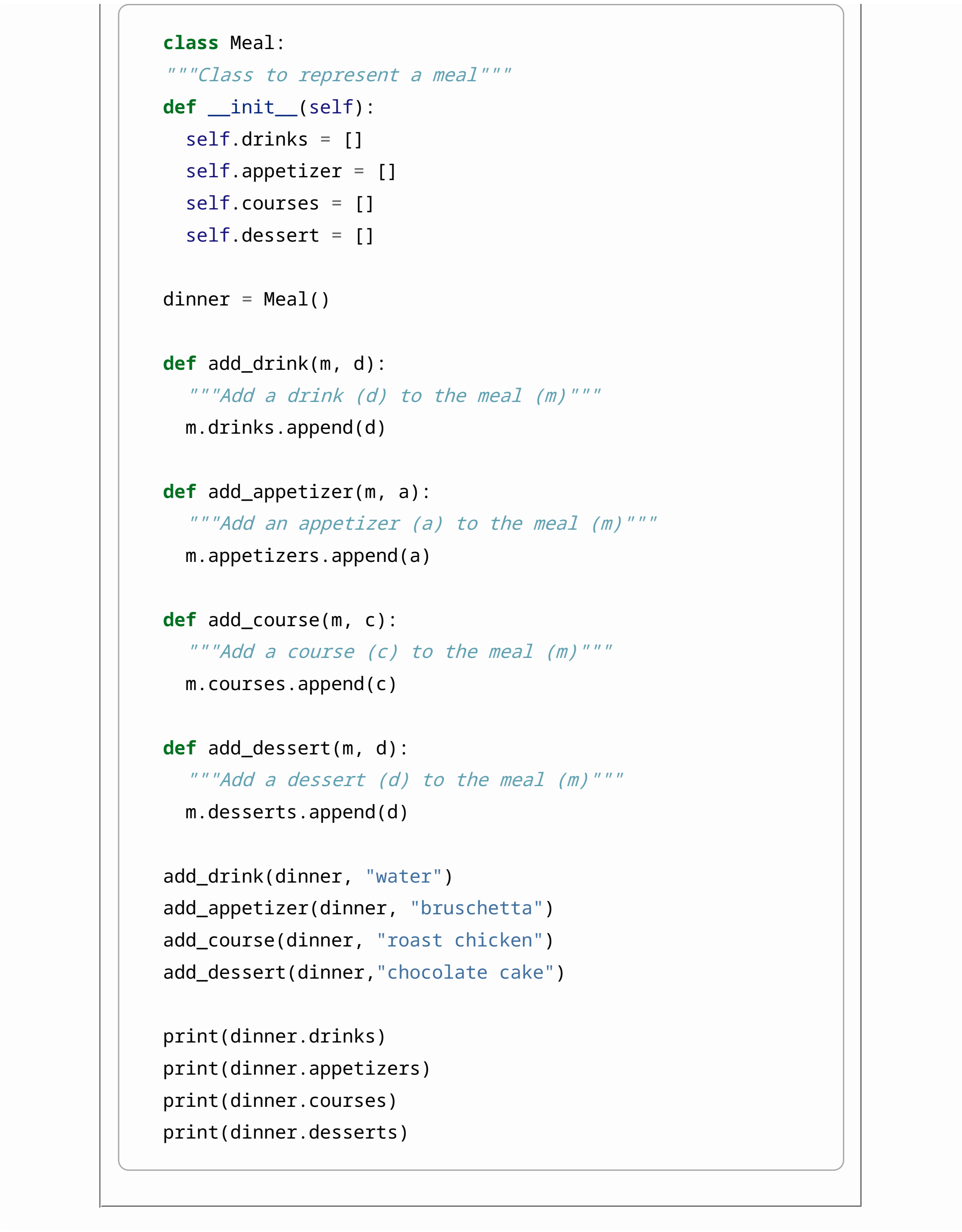
challenge

Create the following methods:

- `add_course` - accepts a string which represents a course and adds it to the meal.
- `add_dessert` - accepts a string which represents a dessert and adds it to the meal.

Test your code using `"roast chicken"` as a main course and `"chocolate cake"` as a dessert. Then print out each course of the meal.

▼ **Meal code**



```
class Meal:
"""Class to represent a meal"""
def __init__(self):
  self.drinks = []
  self.appetizer = []
  self.courses = []
  self.dessert = []

dinner = Meal()

def add_drink(m, d):
  """Add a drink (d) to the meal (m)"""
  m.drinks.append(d)

def add_appetizer(m, a):
  """Add an appetizer (a) to the meal (m)"""
  m.appetizers.append(a)

def add_course(m, c):
  """Add a course (c) to the meal (m)"""
  m.courses.append(c)

def add_dessert(m, d):
  """Add a dessert (d) to the meal (m)"""
  m.desserts.append(d)

add_drink(dinner, "water")
add_appetizer(dinner, "bruschetta")
add_course(dinner, "roast chicken")
add_dessert(dinner,"chocolate cake")

print(dinner.drinks)
print(dinner.appetizers)
print(dinner.courses)
print(dinner.desserts)
```

Printing the Meal 1

Planning the Method

Before writing the method to print the meal, think about what you want the output should like. Imagine that a meal consists of the following courses:

- Drinks - water and coffee
- Appetizers - nothing served as an appetizer
- Main course - roast chicken, mashed potatoes, and salad.
- Dessert - chocolate cake

Change your code to reflect this meal. Also, add the `print_meal` method even though it has not yet been declared.

```
dinner = Meal()
dinner.add_drink("water")
dinner.add_drink("coffee")
dinner.add_course("roast chicken")
dinner.add_course("mashed potatoes")
dinner.add_course("salad")
dinner.add_dessert("chocolate cake")
dinner.print_meal()
```

The `print_meal` method should be able to handle an empty list (nothing served), a list of length 1, a list of length 2, and a list of 3 or more elements. Each of these scenarios has specific requirements: Is the verb singular or plural? Do you need a comma-separated list or just the word `"and"`? Should the word be capitalized?

Nothing was Served

The method should print all of the courses of the meal. So start with a list of all of the courses. Next, iterate over the list. However, we are not going to use the `for course in courses:` syntax. Use `for position in range(4):` syntax. Then create the variable `course` which is assigned the element at index `position`. You will know that nothing is served if the length of `course` is 0.

```
def print_meal(self):
  """Prints the meal"""
  courses = [self.drinks, self.appetizers, self.main_course,
    self.desserts]
  for position in range(4):
   course = courses[position]
   if len(course) == 0: #check for an empty list
```

▼ **Why not use `for course in courses:`?**

Open the Code Visualizer

Later on in this program, you will need access to the index of each element. The list method `index` returns the index of an element in a list. However, this method becomes a problem when you have two or more elements that are the same. Look at the code in the visualizer (click the link above). The loop prints the index of each element. You would expect it to output `0 1 2 3`. Instead it prints `0 1 2 1`. That is because the `index` list method returns the first index with the desired value. So the last element was not accessed by the loop. Using `range` in the for loop allows you to access each element in the list.

Printing a message for an empty list becomes tricky because the sentence changes based on the course.

- No **drinks were** served with the meal.
- No **appetizers were** served with the meal.
- No **main course was** served with the meal.
- No **dessert was** served with the meal.

The best way to handle this is to use an f-string for the print statement and a helper method to generate the text in bold from above.

```
def course_name(self, position):
 pass

def print_meal(self):
  """Prints the meal"""
  courses = [self.drinks, self.appetizers, self.main_course,
    self.desserts]
  for position in range(4):
   course = courses[position]
   if len(course) == 0: #check for an empty list
    print(f"No {self.course_name(position)} served with the
    meal.")
```

The helper method `course_name` will compare `course` with each of the instance attributes and return the required string.

```python
def course_name(self, position):
    if position == 0:
      return "drinks were"
    elif position == 1:
      return "appetizers were"
    elif position == 2:
      return "main course was"
    elif position == 3:
      return "dessert was"
```

▼ **The power of f-strings**

F-strings are more concise than using the `.format` string method. They also allow you to put almost anything between the `{ }` characters. Variables, expressions, function calls, and method calls are all valid with f-strings.

Running the code now should produce `"No appetizers were served with the meal."` since it is the only course that is an empty string.

challenge

Try this variation:

Use the comment symbol # to comment out all of the lines with a method that adds a course to the `dinner` object. Run the program. The output should be:

```
No drinks were served with the meal.
No appetizers were served with the meal.
No main course was served with the meal.
No dessert was served with the meal.
```

▼ **Code**

Here are some examples for the functions suggested above.

```
dinner = Meal()
#dinner.add_drink("water")
#dinner.add_drink("coffee")
#dinner.add_course("roast chicken")
#dinner.add_course("mashed potatoes")
#dinner.add_course("salad")
#dinner.add_dessert("chocolate cake")
dinner.print_meal()
```

Printing the Meal 2

One Item Was Served

This is a simple case. If only one item is served, that item should be capitalized followed by `" was served with the meal."`. Use the `capitalize` method to capitalize the first letter of a string.

```python
def print_meal(self):
    """Prints the meal"""
    courses = [self.drinks, self.appetizers, self.main_course, self.desserts]
    for position in range(4):
      course = courses[position]
      if len(course) == 0: #check for an empty list
        print(f"No {self.course_name(position)} served with the meal.")
      elif len(course) == 1: #check for only one item
        print(f"{course[0].capitalize()} was served with the meal.")
```

Note, remove the comment symbol # for `dinner.add_dessert("chocolate cake")`.

Two Items Were Served

If there are two items being served, the first item should be capitalized followed by `and` and the second item. The sentence will end with `" were served with the meal."`.

```
def print_meal(self):
    """Prints the meal"""
    courses = [self.drinks, self.appetizers, self.main_course,
        self.desserts]
    for position in range(4):
      course = courses[position]
      if len(course) == 0: #check for an empty list
        print(f"No {self.course_name(position)} served with the
        meal.")
      elif len(course) == 1: #check for only one item
        print(f"{course[0].capitalize()} was served with the
        meal.")
      elif len(course) == 2: #check for only two items
        print(f"{course[0].capitalize()} and {course[1]} were
        served with the meal.")
```

Note, remove the comment symbol # for `dinner.add_drink("water")` and `dinner.add_drink("coffee")`.

More than Two Items Were Served

If more than two items are served, then you need a comma-separated list. The first item should be capitalized followed by a comma and a space. The next items are followed by commas and spaces. The final item in the list is prefaced with `and`. No comma is used after the last item. The sentence ends with `" were served with the meal."`. Remember, the final print statement needs to add a new line character. Be sure that it does not have `end=""` in it.

```
def print_meal(self):
    """Prints the meal"""
    courses = [self.drinks, self.appetizers, self.main_course,
        self.desserts]
    for position in range(4):
      course = courses[position]
      if len(course) == 0: #check for an empty list
        print(f"No {self.course_name(position)} served with the
        meal.")
      elif len(course) == 1: #check for only one item
        print(f"{course[0].capitalize()} was served with the
        meal.")
      elif len(course) == 2: #check for only two items
        print(f"{course[0].capitalize()} and {course[1]} were
        served with the meal.")
      else: #many items were served
        for item in course:
          if course.index(item) == 0: #check to see if first
        element
            print(f"{item.capitalize()}, ", end="")
          elif item == course[-1]: #check to see if last element
            print(f"and {item} ", end="")
          else:
            print(f"{item}, ", end="")
        print("were served with the meal.")
```

challenge

Check your work:

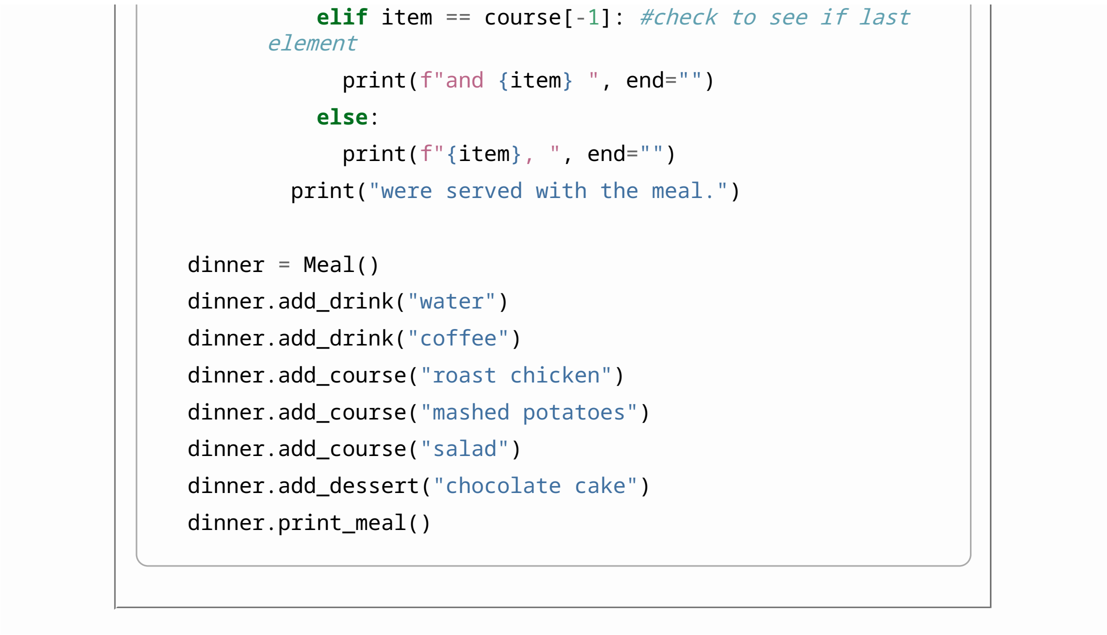
Create different meals and make sure your program works as expected. For example:

```
dinner = Meal()
dinner.add_drink("white wine")
dinner.add_appetizer("tapenade")
dinner.add_appetizer("antipasto")
dinner.add_course("cauliflower bolognese")
dinner.add_course("butternut squash soup")
dinner.add_course("kale salad")
dinner.print_meal()
```

▼ **Meal Code**

```
class Meal:
  """Class to represent a meal"""
```

```
def __init__(self):
self.drinks = []
self.appetizers = []
self.main_course = []
self.desserts = []

def add_drink(self, d):
  """Add a drink (d) to the meal (self)"""
  self.drinks.append(d)

def add_appetizer(self, a):
  """Add an appetizer (a) to the meal (self)"""
  self.appetizers.append(a)

def add_course(self, c):
  self.main_course.append(c)

def add_dessert(self, d):
  self.desserts.append(d)

def course_name(self, position):
  if position == 0:
    return "drinks were"
  elif position == 1:
    return "appetizers were"
  elif position == 2:
    return "main course was"
  elif position == 3:
    return "dessert was"

def print_meal(self):
  """Prints the meal"""
  courses = [self.drinks, self.appetizers, self.main_course, self.desserts]
  for position in range(4):
    course = courses[position]
    if len(course) == 0: #check for an empty list
      print(f"No {self.course_name(position)} served with the meal.")
    elif len(course) == 1: #check for only one item
      print(f"{course[0].capitalize()} was served with the meal.")
    elif len(course) == 2: #check for only two items
      print(f"{course[0].capitalize()} and {course[1]} were served with the meal.")
    else: #many items were served
      for item in course:
        if course.index(item) == 0: #check to see if first element
          print(f"{item.capitalize()}, ", end="")
```



```
          elif item == course[-1]: #check to see if last
      element
            print(f"and {item} ", end="")
          else:
            print(f"{item}, ", end="")
        print("were served with the meal.")

dinner = Meal()
dinner.add_drink("water")
dinner.add_drink("coffee")
dinner.add_course("roast chicken")
dinner.add_course("mashed potatoes")
dinner.add_course("salad")
dinner.add_dessert("chocolate cake")
dinner.print_meal()
```

Changing Objects with Methods Formative Assessment 1

Changing Objects with Methods Formative Assessment 2