Python

2023-12-31

Contents

1 VARIABLES AND SIMPLE DATA TYPES

- Changing Case in a String with Methods

```
# Escribe la primera letra de cada palabra en mayúscula
name = "ada lovelace"
print(name.title())
```

```
## Ada Lovelace
```

```
# Escribe toda la palabra en mayúsculas
print(name.upper())
```

```
## ADA LOVELACE
```

```
# Escribe toda la palabra en minúsculas
print(name.lower())
```

```
## ada lovelace
```

- Using Variables in Strings

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
print(full_name)
```

```
## ada lovelace
```

- Adding Whitespace to Strings with Tabs or Newlines

```
print("Languages:\n\tPython\n\tC\n\tJavaScript")
```

```
## Languages:
##  Python
##  C
##  JavaScript
```

- Stripping Whitespace

```
favorite_language = ' python '
favorite_language.rstrip()
```

```
## ' python'
```

```
favorite_language.lstrip()
```

```
## 'python '
```

```
favorite_language.strip()
```

```
## 'python'
```

- Removing Prefixes

```
nostarch_url = 'https://nostarch.com'
nostarch_url.removeprefix('https://')
```

```
## 'nostarch.com'
```

- Underscores in Numbers

```
universe_age = 14_000_000_000
print(universe_age)
```

```
## 14000000000
```

- Multiple Assignment

  ```
  x, y, z = 0, 0, 0
  ```

2 INTRODUCING LISTS

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

```
## ['trek', 'cannondale', 'redline', 'specialized']
```

- Accessing Elements in a List

  ```
  print(bicycles[0].title())
  ```

  ```
  ## Trek
  ```

  Python has a special syntax for accessing the last element in a list. If you ask for the item at index -1, Python always returns the last item in the list:

  ```
  print(bicycles[-1])
  ```

  ```
  ## specialized
  ```

- Using Individual Values from a List

  ```
  message = f"My first bicycle was a {bicycles[0].title()}."
  print(message)
  ```

  ```
  ## My first bicycle was a Trek.
  ```

- Modifying Elements in a List

  ```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)
  ```

  ```
  ## ['honda', 'yamaha', 'suzuki']
  ```

  ```
  motorcycles[0] = 'ducati'
  print(motorcycles)
  ```

  ```
  ## ['ducati', 'yamaha', 'suzuki']
  ```

- Adding Elements to a List

  ```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)
  ```

  ```
  ## ['honda', 'yamaha', 'suzuki']
  ```

  ```
  motorcycles.append('ducati')
  print(motorcycles)
  ```

  ```
  ## ['honda', 'yamaha', 'suzuki', 'ducati']
  ```

- Inserting Elements into a List

  ```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  motorcycles.insert(0, 'ducati')
  print(motorcycles)
  ```

```
## ['ducati', 'honda', 'yamaha', 'suzuki']
```

- Removing an Item Using the del Statement

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
## ['honda', 'yamaha', 'suzuki']
```

```
del motorcycles[0]
print(motorcycles)
```

```
## ['yamaha', 'suzuki']
```

- Removing an Item Using the pop() Method

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
## ['honda', 'yamaha', 'suzuki']
```

```
popped_motorcycle = motorcycles.pop()
print(motorcycles)
```

```
## ['honda', 'yamaha']
```

```
print(popped_motorcycle)
```

```
## suzuki
```

- Popping Items from Any Position in a List

```
motorcycles = ['honda', 'yamaha', 'suzuki']
first_owned = motorcycles.pop(0)
print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

```
## The first motorcycle I owned was a Honda.
```

- Removing an Item by Value

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
```

```
## ['honda', 'yamaha', 'suzuki', 'ducati']
```

```
motorcycles.remove('ducati')
print(motorcycles)
```

```
## ['honda', 'yamaha', 'suzuki']
```

- Sorting a List Permanently with the sort() Method

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

```
## ['audi', 'bmw', 'subaru', 'toyota']
```

- Sorting a List Temporarily with the sorted() Function

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print("Here is the original list:")
```

```
## Here is the original list:
```

```
print(cars)
```

```
## ['bmw', 'audi', 'toyota', 'subaru']
```

```
print("\nHere is the sorted list:")
```

```
##
## Here is the sorted list:
```

```
print(sorted(cars))
```

```
## ['audi', 'bmw', 'subaru', 'toyota']
```

```
print("\nHere is the original list again:")
```

```
##
## Here is the original list again:
```

```
print(cars)
```

```
## ['bmw', 'audi', 'toyota', 'subaru']
```

- Printing a List in Reverse Order

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)
```

```
## ['bmw', 'audi', 'toyota', 'subaru']
```

```
cars.reverse()
print(cars)
```

```
## ['subaru', 'toyota', 'audi', 'bmw']
```

- Finding the Length of a List

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
len(cars)
```

```
## 4
```

3 WORKING WITH LISTS

- Looping Through an Entire List

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
  print(magician)
```

```
## alice
## david
## carolina
```

- Using the range() Function

```
for value in range(1, 5):
  print(value)
```

```
## 1
## 2
## 3
## 4
```

- Using range() to Make a List of Numbers

```
numbers = list(range(1, 6))
print(numbers)
```

```
## [1, 2, 3, 4, 5]
```

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

```
## [2, 4, 6, 8, 10]
```

```
squares = []
for value in range(1,11):
  squares.append(value**2)
print(squares)
```

```
## [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Simple Statistics with a List of Numbers

```
digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
min(digits)
```

```
## 0
```

```
max(digits)
```

```
## 9
```

```
sum(digits)
```

```
## 45
```

- List Comprehensions

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

```
## [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Slicing a List

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
```

```
## ['charles', 'martina', 'michael']
```

```
print(players[:4])
```

```
## ['charles', 'martina', 'michael', 'florence']
```

```
print(players[2:])
```

```
## ['michael', 'florence', 'eli']
```

```
# con el signo negativo cuenta desde el final
print(players[-3:])
```

```
## ['michael', 'florence', 'eli']
```

- Looping Through a Slice

```
print("Here are the first three players on my team:")
```

```
## Here are the first three players on my team:
```

```
for player in players[:3]:
  print(player.title())
```

```
## Charles
## Martina
## Michael
```

- Copying a List

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]
print("My favorite foods are:")
```

```
## My favorite foods are:
```

```
print(my_foods)
```

```
## ['pizza', 'falafel', 'carrot cake']
```

```
print("\nMy friend's favorite foods are:")
```

```
##
## My friend's favorite foods are:
```

```
print(friend_foods)
```

```
## ['pizza', 'falafel', 'carrot cake']
```

- Defining a Tuple Las tuplas son como las listas, pero no se pueden modificar. Si queremos cambiar una tupla tenemos que redefinirla.

```
dimensions = (200, 50)
print(dimensions[0])
```

```
## 200
```

```
print(dimensions[1])
```

```
## 50
```

4 IF STATEMENTS

- A Simple Example

```
cars = ['audi', 'bmw', 'subaru', 'toyota']
for car in cars:
  if car == 'bmw':
    print(car.upper())
  else:
    print(car.title())
```

```
## Audi
## BMW
## Subaru
## Toyota
```

- Checking for Inequality

```
requested_topping = 'mushrooms'
if requested_topping != 'anchovies':
  print("Hold the anchovies!")
```

```
## Hold the anchovies!
```

- Numerical Comparisons

```
answer = 17
if answer != 42:
  print("That is not the correct answer. Please try again!")
```

```
## That is not the correct answer. Please try again!
```

- Checking Whether a Value Is in a List

```
requested_toppings = ['mushrooms', 'onions', 'pineapple']
'mushrooms' in requested_toppings
```

```
## True
```

```
'pepperoni' in requested_toppings
```

```
## False
```

- Checking Whether a Value Is Not in a List

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
if user not in banned_users:
  print(f"{user.title()}, you can post a response if you wish.")
```

```
## Marie, you can post a response if you wish.
```

- Testing Multiple Conditions The if- elif- else block would stop running after only one test passes.
- Using if Statements with Lists

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']
for requested_topping in requested_toppings:
  print(f"Adding {requested_topping}.")
```

```
## Adding mushrooms.
## Adding green peppers.
## Adding extra cheese.
```

```
print("\nFinished making your pizza!")
```

```
##
## Finished making your pizza!
```

```
for requested_topping in requested_toppings:
  if requested_topping == 'green peppers':
    print("Sorry, we are out of green peppers right now.")
  else:
    print(f"Adding {requested_topping}.")
```

```
## Adding mushrooms.
## Sorry, we are out of green peppers right now.
## Adding extra cheese.
```

```
print("\nFinished making your pizza!")
```

```
##
## Finished making your pizza!
```

- Checking That a List Is Not Empty

```
requested_toppings = []
if requested_toppings:
  for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")
  print("\nFinished making your pizza!")
else:
  print("Are you sure you want a plain pizza?")
```

```
## Are you sure you want a plain pizza?
```

- Using Multiple Lists

```
available_toppings = ['mushrooms', 'olives', 'green peppers',
'pepperoni', 'pineapple', 'extra cheese']
requested_toppings = ['mushrooms', 'french fries', 'extra cheese']
for requested_topping in requested_toppings:
  if requested_topping in available_toppings:
    print(f"Adding {requested_topping}.")
  else:
    print(f"Sorry, we don't have {requested_topping}.")
```

```
## Adding mushrooms.
## Sorry, we don't have french fries.
## Adding extra cheese.
```

```
print("\nFinished making your pizza!")
```

```
##
## Finished making your pizza!
```

5 DICTIONARIES

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0['color'])
```

```
## green
```

```
print(alien_0['points'])
```

```
## 5
```

- Adding New Key-Value Pairs

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)
```

```
## {'color': 'green', 'points': 5}
```

```
alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

```
## {'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}
```

- Starting with an Empty Dictionary

```python
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
print(alien_0)
```

```
## {'color': 'green', 'points': 5}
```

- Modifying Values in a Dictionary

```python
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")
```

```
## The alien is green.
```

```python
alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

```
## The alien is now yellow.
```

- Removing Key-Value Pairs

```python
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)
```

```
## {'color': 'green', 'points': 5}
```

```python
del alien_0['points']
print(alien_0)
```

```
## {'color': 'green'}
```

- Using get() to Access Values

  Using keys in square brackets to retrieve the value you're interested in from a dictionary might cause one potential problem: if the key you ask for doesn't exist, you'll get an error.

  The get() method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn't exist:

```python
alien_0 = {'color': 'green', 'speed': 'slow'}
point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

```
## No point value assigned.
```

- Looping Through a Dictionary

```python
user_0 = {
  'username': 'efermi',
  'first': 'enrico',
  'last': 'fermi',
}

for key, value in user_0.items():
  print(f"\nKey: {key}")
  print(f"Value: {value}")
```

```
##
## Key: username
## Value: efermi
##
## Key: first
## Value: enrico
##
## Key: last
## Value: fermi
```

- Looping Through All the Keys in a Dictionary

```
favorite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'rust',
  'phil': 'python',
}

for name in favorite_languages.keys():
  print(name.title())
```

```
## Jen
## Sarah
## Edward
## Phil
```

Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote:

```
for name in favorite_languages:
  print(name.title())
```

```
## Jen
## Sarah
## Edward
## Phil
```

You can access the value associated with any key you care about inside the loop, by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favorite language:

```
friends = ['phil', 'sarah']
for name in favorite_languages.keys():
  print(f"Hi {name.title()}.")

  if name in friends:
    language = favorite_languages[name].title()
    print(f"\t{name.title()}, I see you love {language}!")
```

```
## Hi Jen.
## Hi Sarah.
##  Sarah, I see you love C!
## Hi Edward.
## Hi Phil.
##  Phil, I see you love Python!
```

You can also use the keys() method to find out if a particular person was polled. This time, let's find out if Erin took the poll:

```python
if 'erin' not in favorite_languages.keys():
  print("Erin, please take our poll!")
```

```
## Erin, please take our poll!
```

- Looping Through a Dictionary's Keys in a Particular Order

```python
for name in sorted(favorite_languages.keys()):
  print(f"{name.title()}, thank you for taking the poll.")
```

```
## Edward, thank you for taking the poll.
## Jen, thank you for taking the poll.
## Phil, thank you for taking the poll.
## Sarah, thank you for taking the poll.
```

- Looping Through All Values in a Dictionary

```python
print("The following languages have been mentioned:")
```

```
## The following languages have been mentioned:
```

```python
for language in favorite_languages.values():
  print(language.title())
```

```
## Python
## C
## Rust
## Python
```

This approach pulls all the values from the dictionary without checking for repeats. This might work fine with a small number of values, but in a poll with a large number of respondents, it would result in a very repetitive list. To see each language chosen without repetition, we can use a set. A set is a collection in which each item must be unique:

```python
print("The following languages have been mentioned:")
```

```
## The following languages have been mentioned:
```

```python
for language in set(favorite_languages.values()):
  print(language.title())
```

```
## Python
## Rust
## C
```

You can build a set directly using braces and separating the elements with commas:

```python
languages = {'python', 'rust', 'python', 'c'}
languages
```

```
## {'python', 'rust', 'c'}
```

- A List of Dictionaries

```python
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}
aliens = [alien_0, alien_1, alien_2]
for alien in aliens:
  print(alien)
```

```
## {'color': 'green', 'points': 5}
## {'color': 'yellow', 'points': 10}
## {'color': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example, we use range() to create a fleet of 30 aliens:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range(30):
  new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
  aliens.append(new_alien)

# Show the first 5 aliens.
for alien in aliens[:5]:
  print(alien)
```

```
## {'color': 'green', 'points': 5, 'speed': 'slow'}
## {'color': 'green', 'points': 5, 'speed': 'slow'}
## {'color': 'green', 'points': 5, 'speed': 'slow'}
## {'color': 'green', 'points': 5, 'speed': 'slow'}
## {'color': 'green', 'points': 5, 'speed': 'slow'}
```

```
print("...")
```

```
## ...
```

```
# Show how many aliens have been created.
print(f"Total number of aliens: {len(aliens)}")
```

```
## Total number of aliens: 30
```

We can use a for loop and an if statement to change the color of the aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (30):
  new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
  aliens.append(new_alien)

for alien in aliens[:3]:
  if alien['color'] == 'green':
    alien['color'] = 'yellow'
    alien['speed'] = 'medium'
    alien['points'] = 10

# Show the first 5 aliens.
for alien in aliens[:5]:
  print(alien)
```

```
## {'color': 'yellow', 'points': 10, 'speed': 'medium'}
## {'color': 'yellow', 'points': 10, 'speed': 'medium'}
## {'color': 'yellow', 'points': 10, 'speed': 'medium'}
## {'color': 'green', 'points': 5, 'speed': 'slow'}
## {'color': 'green', 'points': 5, 'speed': 'slow'}
```

```
print("...")
```

```
## ...
```

- A List in a Dictionary

```
# Store information about a pizza being ordered.
pizza = {
  'crust': 'thick',
  'toppings': ['mushrooms', 'extra cheese'],
}

# Summarize the order.
print(f"You ordered a {pizza['crust']}-crust pizza "
  "with the following toppings:")
```

```
## You ordered a thick-crust pizza with the following toppings:
```

```
for topping in pizza['toppings']:
  print(f"\t{topping}")
```

```
##  mushrooms
##  extra cheese
```

You can nest a list inside a dictionary anytime you want more than one value to be associated with a single key in a dictionary.

```
favorite_languages = {
  'jen': ['python', 'rust'],
  'sarah': ['c'],
  'edward': ['rust', 'go'],
  'phil': ['python', 'haskell'],
}

for name, languages in favorite_languages.items():
  print(f"\n{name.title()}'s favorite languages are:")
  for language in languages:
    print(f"\t{language.title()}")
```

```
##
## Jen's favorite languages are:
##  Python
##  Rust
##
## Sarah's favorite languages are:
##  C
##
## Edward's favorite languages are:
##  Rust
##  Go
##
## Phil's favorite languages are:
##  Python
##  Haskell
```

- A Dictionary in a Dictionary

```
users = {
  'aeinstein': {
    'first': 'albert',
    'last': 'einstein',
    'location': 'princeton',
  },

  'mcurie': {
    'first': 'marie',
    'last': 'curie',
    'location': 'paris',
  },
}

for username, user_info in users.items():
  print(f"\nUsername: {username}")
  full_name = f"{user_info['first']} {user_info['last']}"
  location = user_info['location']

  print(f"\tFull name: {full_name.title()}")
  print(f"\tLocation: {location.title()}")
```

```
##
## Username: aeinstein
##  Full name: Albert Einstein
##  Location: Princeton
##
## Username: mcurie
##  Full name: Marie Curie
##  Location: Paris
```

6 USER INPUT AND WHILE LOOPS

- How the input() Function Works

```
# message = input("Tell me something, and I will repeat it back to you: ")
# print(message)
```

- Writing Clear Prompts

```
# name = input("Please enter your name: ")
#print(f"\nHello, {name}!")
```

Sometimes you'll want to write a prompt that's longer than one line.

```
# prompt = "If you share your name, we can personalize the messages you see."
# prompt += "\nWhat is your first name? "

# name = input(prompt)
# print(f"\nHello, {name}!")
```

- Using int() to Accept Numerical Input

```
# height = input("How tall are you, in inches? ")
# height = int(height)
# if height >= 48:
  # print("\nYou're tall enough to ride!")
# else:
  # print("\nYou'll be able to ride when you're a little older.")
```

- The Modulo Operator

```
# number = input("Enter a number, and I'll tell you if it's even or odd: ")
# number = int(number)

# if number % 2 == 0:
  # print(f"\nThe number {number} is even.")
# else:
  # print(f"\nThe number {number} is odd.")
```

- The while Loop in Action

```
current_number = 1
while current_number <= 5:
  print(current_number)
  current_number += 1
```

```
## 1
## 2
## 3
## 4
## 5
```

- Letting the User Choose When to Quit

```
# prompt = "\nTell me something, and I will repeat it back to you:"
# prompt += "\nEnter 'quit' to end the program. "

# message = ""
# while message != 'quit':
  # message = input(prompt)

  # if message != 'quit':
    # print(message)
```

- Using a Flag

```
# active = True
# while active:
  # message = input(prompt)

  # if message == 'quit':
    # active = False
  # else:
    #print(message)
```

- Using break to Exit a Loop

```
# prompt = "\nPlease enter the name of a city you have visited:"
# prompt += "\n(Enter 'quit' when you are finished.) "

# while True:
  # city = input(prompt)

  # if city == 'quit':
    # break
  # else:
    # print(f"I'd love to go to {city.title()}!")
```

- Using continue in a Loop

```
current_number = 0
while current_number < 10:
  current_number += 1
  if current_number % 2 == 0:
    continue
  print(current_number)
```

```
## 1
## 3
## 5
## 7
## 9
```

- Using a while Loop with Lists and Dictionaries
- Moving Items from One List to Another

```
# Start with users that need to be verified,
# and an empty list to hold confirmed users.
unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
# Move each verified user into the list of confirmed users.
while unconfirmed_users:
  current_user = unconfirmed_users.pop()

  print(f"Verifying user: {current_user.title()}")
  confirmed_users.append(current_user)
```

```
## Verifying user: Candace
## Verifying user: Brian
## Verifying user: Alice
```

```
# Display all confirmed users.
print("\nThe following users have been confirmed:")
```

```
##
## The following users have been confirmed:
```

```
for confirmed_user in confirmed_users:
  print(confirmed_user.title())
```

```
## Candace
## Brian
## Alice
```

- Removing All Instances of Specific Values from a List

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)
```

```
## ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
```

```
while 'cat' in pets:
  pets.remove('cat')

print(pets)
```

```
## ['dog', 'dog', 'goldfish', 'rabbit']
```

- Filling a Dictionary with User Input

```
# responses = {}
# # Set a flag to indicate that polling is active.
# polling_active = True

# while polling_active:
  # # Prompt for the person's name and response.
  # name = input("\nWhat is your name? ")
  # response = input("Which mountain would you like to climb someday? ")

  # # Store the response in the dictionary.
  # responses[name] = response

  # # Find out if anyone else is going to take the poll.
  # repeat = input("Would you like to let another person respond? (yes/ no) ")
  # if repeat == 'no':
    # polling_active = False

# # Polling is complete. Show the results.
# print("\n--- Poll Results ---")
# for name, response in responses.items():
  # print(f"{name} would like to climb {response}.")
```

7 FUNCTIONS

- Defining a Function

```
def greet_user():
  """Display a simple greeting."""
  print("Hello!")

greet_user()
```

```
## Hello!
```

- Passing Information to a Function

```
def greet_user(username):
  """Display a simple greeting."""
  print(f"Hello, {username.title()}!")

greet_user('jesse')
```

```
## Hello, Jesse!
```

- Passing Arguments

```
def describe_pet(animal_type, pet_name):
  """Display information about a pet."""
  print(f"\nI have a {animal_type}.")
  print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
```

```
##
## I have a hamster.
## My hamster's name is Harry.
```

- Keyword Arguments

```
def describe_pet(animal_type, pet_name):
  """Display information about a pet."""
  print(f"\nI have a {animal_type}.")
  print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

```
##
## I have a hamster.
## My hamster's name is Harry.
```

- Default Values

  Note that the order of the parameters in the function definition had to be changed. Because the default value makes it unnecessary to specify a type of animal as an argument, the only argument left in the function call is the pet's name. Python still interprets this as a positional argument, so if the function is called with just a pet's name, that argument will match up with the first parameter listed in the function's definition. This is the reason the first parameter needs to be pet_name.

```
def describe_pet(pet_name, animal_type='dog'):
  """Display information about a pet."""
  print(f"\nI have a {animal_type}.")
  print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

```
##
## I have a dog.
## My dog's name is Willie.
```

```
# También se puede hacer así:
describe_pet('willie')
```

```
##
## I have a dog.
## My dog's name is Willie.
```

- Return Values

```
def get_formatted_name(first_name, last_name):
  """Return a full name, neatly formatted."""
  full_name = f"{first_name} {last_name}"
  return full_name.title()
```

```python
musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

```
## Jimi Hendrix
```

- Making an Argument Optional

```python
def get_formatted_name(first_name, last_name, middle_name=''):
  """Return a full name, neatly formatted."""
  if middle_name:
    full_name = f"{first_name} {middle_name} {last_name}"
  else:
    full_name = f"{first_name} {last_name}"
  return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

```
## Jimi Hendrix
```

```python
musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

```
## John Lee Hooker
```

- Returning a Dictionary

```python
def build_person(first_name, last_name):
  """Return a dictionary of information about a person."""
  person = {'first': first_name, 'last': last_name}
  return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

```
## {'first': 'jimi', 'last': 'hendrix'}
```

You can easily extend this function to accept optional values like a middle name, an age, an occupation, or any other information you want to store about a person. For example, the following change allows you to store a person's age as well:

```python
def build_person(first_name, last_name, age=None):
  """Return a dictionary of information about a person."""
  person = {'first': first_name, 'last': last_name}
  if age:
    person['age'] = age
  return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

```
## {'first': 'jimi', 'last': 'hendrix', 'age': 27}
```

- Using a Function with a while Loop

```python
# def get_formatted_name(first_name, last_name):
  # """Return a full name, neatly formatted."""
  # full_name = f"{first_name} {last_name}"
  # return full_name.title()
# while True:
  # print("\nPlease tell me your name:")
  # print("(enter 'q' at any time to quit)")

  # f_name = input("First name: ")
  # if f_name == 'q':
    # break

  # l_name = input("Last name: ")
  # if l_name == 'q':
    # break

  # formatted_name = get_formatted_name(f_name, l_name)
  # print(f"\nHello, {formatted_name}!")
```

- Passing a List

```
def greet_users(names):
  """Print a simple greeting to each user in the list."""
  for name in names:
    msg = f"Hello, {name.title()}!"
    print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

```
## Hello, Hannah!
## Hello, Ty!
## Hello, Margot!
```

- Modifying a List in a Function

```
# Start with some designs that need to be printed.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simulate printing each design, until none are left.
# Move each design to completed_models after printing.
while unprinted_designs:
  current_design = unprinted_designs.pop()
  print(f"Printing model: {current_design}")
  completed_models.append(current_design)
```

```
## Printing model: dodecahedron
## Printing model: robot pendant
## Printing model: phone case
```

```
# Display all completed models.
print("\nThe following models have been printed:")
```

```
##
## The following models have been printed:
```

```
for completed_model in completed_models:
  print(completed_model)
```

```
## dodecahedron
## robot pendant
## phone case
```

We can reorganize this code by writing two functions, each of which does one specific job. Most of the code won't change; we're just structuring it more carefully. The first function will handle printing the designs, and the second will summarize the prints that have been made:

```python
def print_models(unprinted_designs, completed_models):
  """
  Simulate printing each design, until none are left.
  Move each design to completed_models after printing.
  """

  while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

def show_completed_models(completed_models):
  """Show all the models that were printed."""
  print("\nThe following models have been printed:")
  for completed_model in completed_models:
    print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
```

```
## Printing model: dodecahedron
## Printing model: robot pendant
## Printing model: phone case
```

```python
show_completed_models(completed_models)
```

```
##
## The following models have been printed:
## dodecahedron
## robot pendant
## phone case
```

- Preventing a Function from Modifying a List

  Sometimes you'll want to prevent a function from modifying a list. For example, say that you start with a list of unprinted designs and write a function to move them to a list of completed models, as in the previous example. You may decide that even though you've printed all the designs, you want to keep the original list of unprinted designs for your records. But because you moved all the design names out of unprinted_designs, the list is now empty, and the empty list is the only version you have; the original is gone. In this case, you can address this issue by passing the function a copy of the list, not the original. Any changes the function makes to the list will affect only the copy, leaving the original list intact.

  You can send a copy of a list to a function like this:

  ```python
  # function_name(list_name[:])
  ```

  The slice notation [:] makes a copy of the list to send to the function. If we didn't want to empty the list of unprinted designs in printing_models.py, we could call print_models() like this:

```
# print_models(unprinted_designs[:], completed_models)
```

- Passing an Arbitrary Number of Arguments

```
def make_pizza(*toppings):
  """Summarize the pizza we are about to make."""
  print("\nMaking a pizza with the following toppings:")
  for topping in toppings:
    print(f"- {topping}")

make_pizza('pepperoni')
```

```
##
## Making a pizza with the following toppings:
## - pepperoni
```

```
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
##
## Making a pizza with the following toppings:
## - mushrooms
## - green peppers
## - extra cheese
```

- Mixing Positional and Arbitrary Arguments

  If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

  For example, if the function needs to take in a size for the pizza, that parameter must come before the parameter *toppings:

```
def make_pizza(size, *toppings):
  """Summarize the pizza we are about to make."""
  print(f"\nMaking a {size}-inch pizza with the following toppings:")
  for topping in toppings:
    print(f"- {topping}")

make_pizza(16, 'pepperoni')
```

```
##
## Making a 16-inch pizza with the following toppings:
## - pepperoni
```

```
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
##
## Making a 12-inch pizza with the following toppings:
## - mushrooms
## - green peppers
## - extra cheese
```

- Using Arbitrary Keyword Arguments

  Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive. The function build_profile() in the following example always takes in a first and last name, but it accepts an arbitrary number of keyword arguments as well:

```python
def build_profile(first, last, **user_info):
  """Build a dictionary containing everything we know about a user."""
  user_info['first_name'] = first
  user_info['last_name'] = last
  return user_info

user_profile = build_profile('albert', 'einstein',
                              location='princeton',
                              field='physics')
print(user_profile)
```

```
## {'location': 'princeton', 'field': 'physics', 'first_name': 'albert', 'last_name': 'einstein'}
```

The definition of build_profile() expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want. The double asterisks before the parameter **user_info cause Python to create a dictionary called user_info containing all the extra name-value pairs the function receives. Within the function, you can access the key-value pairs in user_info just as you would for any dictionary.

In the body of build_profile(), we add the first and last names to the user_info dictionary because we'll always receive these two pieces of information from the user, and they haven't been placed into the dictionary yet. Then we return the user_info dictionary to the function call line.

We call build_profile(), passing it the first name 'albert', the last name 'einstein', and the two key-value pairs location='princeton' and field='physics'. We assign the returned profile to user_profile and print user_profile.

- Storing Your Functions in Modules
- Importing an Entire Module

  To start importing functions, we first need to create a module. A module is a file ending in .py that contains the code you want to import into your program.

  ```
  file pizza.py

  def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
      print(f"- {topping}")
  ```

  Now we'll make a separate file called making_pizzas.py in the same directory as pizza.py. This file imports the module we just created and then makes two calls to make_pizza():

  ```
  file making_pizzas.py

  import pizza

  pizza.make_pizza(16, 'pepperoni')
  pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
  ```

  To call a function from an imported module, enter the name of the module you imported, pizza, followed by the name of the function, make_pizza(), separated by a dot.

- Importing Specific Functions

  ```
  from module_name import function_name
  ```

You can import as many functions as you want from a module by separating each function's name with a comma:

```
from module_name import function_0, function_1, function_2
```

With this syntax, you don't need to use the dot notation when you call a function. Because we've explicitly imported the function make_pizza() in the import statement, we can call it by name when we use the function.

- Using as to Give a Function an Alias

  Here we give the function make_pizza() an alias, mp(), by importing make _pizza as mp. The as keyword renames a function using the alias you provide:

  ```
  from pizza import make_pizza as mp

  mp(16, 'pepperoni')
  mp(12, 'mushrooms', 'green peppers', 'extra cheese')
  ```

  The general syntax for providing an alias is:

  ```
  from module_name import function_name as fn
  ```

- Using as to Give a Module an Alias

  ```
  import pizza as p

  p.make_pizza(16, 'pepperoni')
  p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
  ```

  The general syntax for this approach is:

  ```
  import module_name as mn
  ```

- Importing All Functions in a Module

  ```
  from pizza import *

  make_pizza(16, 'pepperoni')
  make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
  ```

  Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get unexpected results.

- Styling Functions

  If you specify a default value for a parameter, no spaces should be used on either side of the equal sign.

  ```
  def function_name(parameter_0, parameter_1='default value')
  ```

  The same convention should be used for keyword arguments in function calls:

  ```
  function_name(value_0, parameter_1='value')
  ```

  If a set of parameters causes a function's definition to be longer than 79 characters, press ENTER after the opening parenthesis on the definition line. On the next line, press the TAB key twice to separate the list of arguments from the body of the function, which will only be indented one level.

  "' def function_name(parameter_0, parameter_1, parameter_2, parameter_3, parameter_4, parameter_5): function body…

8 CLASSES

- Creating the Dog Class

```python
class Dog:
  """A simple attempt to model a dog."""

  def __init__(self, name, age):
    """Initialize name and age attributes."""
    self.name = name
    self.age = age

  def sit(self):
    """Simulate a dog sitting in response to a command."""
    print(f"{self.name} is now sitting.")

  def roll_over(self):
    """Simulate rolling over in response to a command."""
    print(f"{self.name} rolled over!")
```

- Making an Instance from a Class

```python
my_dog = Dog('Willie', 6)

print(f"My dog's name is {my_dog.name}.")
```

```
## My dog's name is Willie.
```

```python
print(f"My dog is {my_dog.age} years old.")
```

```
## My dog is 6 years old.
```

- Working with Classes and Instances
- Setting a Default Value for an Attribute

```python
class Car:

  def __init__(self, make, model, year):
    """Initialize attributes to describe a car."""
    self.make = make
    self.model = model
    self.year = year
    self.odometer_reading = 0

  def get_descriptive_name(self):
    """Return a neatly formatted descriptive name."""
    long_name = f"{self.year} {self.make} {self.model}"
    return long_name.title()

  def read_odometer(self):
    """Print a statement showing the car's mileage."""
    print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
## 2024 Audi A4
```

```python
my_new_car.read_odometer()
```

```
## This car has 0 miles on it.
```

- Modifying an Attribute's Value Directly

```python
my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
## 2024 Audi A4
```

```python
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
## This car has 23 miles on it.
```

- Modifying an Attribute's Value Through a Method

```python
class Car:

  def __init__(self, make, model, year):
    """Initialize attributes to describe a car."""
    self.make = make
    self.model = model
    self.year = year
    self.odometer_reading = 0

  def get_descriptive_name(self):
    """Return a neatly formatted descriptive name."""
    long_name = f"{self.year} {self.make} {self.model}"
    return long_name.title()

  def read_odometer(self):
    """Print a statement showing the car's mileage."""
    print(f"This car has {self.odometer_reading} miles on it.")

  def update_odometer(self, mileage):
    """
    Set the odometer reading to the given value.
    Reject the change if it attemps to roll the odometer back.
    """
    if mileage >= self.odometer_reading:
      self.odometer_reading = mileage
    else:
      print("You can't roll back an odometer!")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
## 2024 Audi A4
```

```python
my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

```
## This car has 23 miles on it.
```

- Incrementing an Attribute's Value Through a Method

```python
class Car:

  def __init__(self, make, model, year):
    """Initialize attributes to describe a car."""
    self.make = make
    self.model = model
    self.year = year
    self.odometer_reading = 0

  def get_descriptive_name(self):
    """Return a neatly formatted descriptive name."""
    long_name = f"{self.year} {self.make} {self.model}"
    return long_name.title()

  def read_odometer(self):
    """Print a statement showing the car's mileage."""
    print(f"This car has {self.odometer_reading} miles on it.")

  def update_odometer(self, mileage):
    """
    Set the odometer reading to the given value.
    Reject the change if it attemps to roll the odometer back.
    """
    if mileage >= self.odometer_reading:
      self.odometer_reading = mileage
    else:
      print("You can't roll back an odometer!")

  def increment_odometer(self, miles):
    """Add the given amount to the odometer reading."""
    self.odometer_reading += miles

my_user_car = Car('subaru', 'outback', 2019)
print(my_user_car.get_descriptive_name())
```

```
## 2019 Subaru Outback
```

```python
my_user_car.update_odometer(23_500)
my_user_car.read_odometer()
```

```
## This car has 23500 miles on it.
```

```python
my_user_car.increment_odometer(100)
my_user_car.read_odometer()
```

```
## This car has 23600 miles on it.
```

- Inheritance
- The ___init___() Method for a Child Class

```python
class Car:
  """A simple attempt to represent a car."""

  def __init__(self, make, model, year):
    """Initialize attributes to describe a car."""
    self.make = make
    self.model = model
    self.year = year
    self.odometer_reading = 0

  def get_descriptive_name(self):
    """Return a neatly formatted descriptive name."""
    long_name = f"{self.year} {self.make} {self.model}"
    return long_name.title()

  def read_odometer(self):
    """Print a statement showing the car's mileage."""
    print(f"This car has {self.odometer_reading} miles on it.")

  def update_odometer(self, mileage):
    """Set the odometer reading to the given value."""
    if mileage >= self.odometer_reading:
      self.odometer_reading = mileage
    else:
      print("You can't roll back an odometer!")

  def increment_odometer(self, miles):
    """Add the given amount to the odometer reading."""
    self.odometer_reading += miles

class ElectricCar(Car):
  """Represent aspects of a car, specific to electric vehicles."""

  def __init__(self, make, model, year):
    """Initialize attributes of the parent class."""
    super().__init__(make, model, year)

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

```
## 2024 Nissan Leaf
```

- Defining Attributes and Methods for the Child Class

```
class ElectricCar(Car):
  """Represent aspects of a car, specific to electric vehicles."""

  def __init__(self, make, model, year):
    """
    Initialize attributes of the parent class.
    Then initialize attributes specific to an electric car.
    """

    super().__init__(make, model, year)
    self.battery_size = 40

  def describe_battery(self):
    """Print a statement describing the battery size."""
    print(f"This car has a {self.battery_size}-kWh battery.")

my_leaf = ElectricCar('nissan', 'leaf', 2024)
```

```
print(my_leaf.get_descriptive_name())
```

```
## 2024 Nissan Leaf
```

```
my_leaf.describe_battery()
```

```
## This car has a 40-kWh battery.
```

- Overriding Methods from the Parent Class

  You define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

  ```
  class ElectricCar(Car):
  --snip--

  def fill_gas_tank(self):
    """Electric cars don't have gas tanks."""
    print("This car doesn't have a gas tank!")
  ```

- Instances as Attributes

  You can break your large class into smaller classes that work together; this approach is called composition.

  we can move those attributes and methods to a separate class called Battery. Then we can use a Battery instance as an attribute in the ElectricCar class:

  ```
  class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=40):
      """Initialize the battery's attributes."""
      self.battery_size = battery_size

    def describe_battery(self):
      """Print a statement describing the battery size."""
      print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
      """Print a statement about the range this battery provides."""
      if self.battery_size == 40:
        range = 150
      elif self.battery_size == 65:
        range = 225

      print(f"This car can go about {range} miles on a full charge.")

  class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
      """
      Initialize attributes of the parent class.
      Then initialize attributes specific to an electric car.
      """

      super().__init__(make, model, year)
  ```

```python
    self.battery = Battery()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

```
## 2024 Nissan Leaf
```

```python
my_leaf.battery.describe_battery()
```

```
## This car has a 40-kWh battery.
```

```python
my_leaf.battery.get_range()
```

```
## This car can go about 150 miles on a full charge.
```