


- **Define and Call a Function**
- **Import and Use a Module**
- **Define and Instantiate a Class**



```
def questions(name, quest, fav_color):  
    print("Your name is {}".format(name))  
    print("Your quest is {}".format(quest))  
    print("Your favorite color is {}".format(fav_color))  
  
questions("Lancelot", "to seek the Holy Grail", "blue")
```





```
def questions(name, quest, fav_color):  
    """Parrot user's answers back to them."""  
    print("Your name is {}".format(name))  
    print("Your quest is {}".format(quest))  
    print("Your favorite color is {}".format(fav_color))
```

```
def age(year_of_birth, current_year):  
    return current_year - year_of_birth
```

```
>>> john_age = age(1939, 2015)  
>>> print("John is approximately {} years old.".format(john_age))  
John is approximately 76 years old.
```



```
def questions(name, quest, favorite_color):  
    print("Your name is {}".format(name))  
    print("Your quest is {}".format(quest))  
    print("Your favorite color is {}".format(favorite_color))  
    name_prefix = "Sir" + name
```

```
name_prefix = "Sir"
```

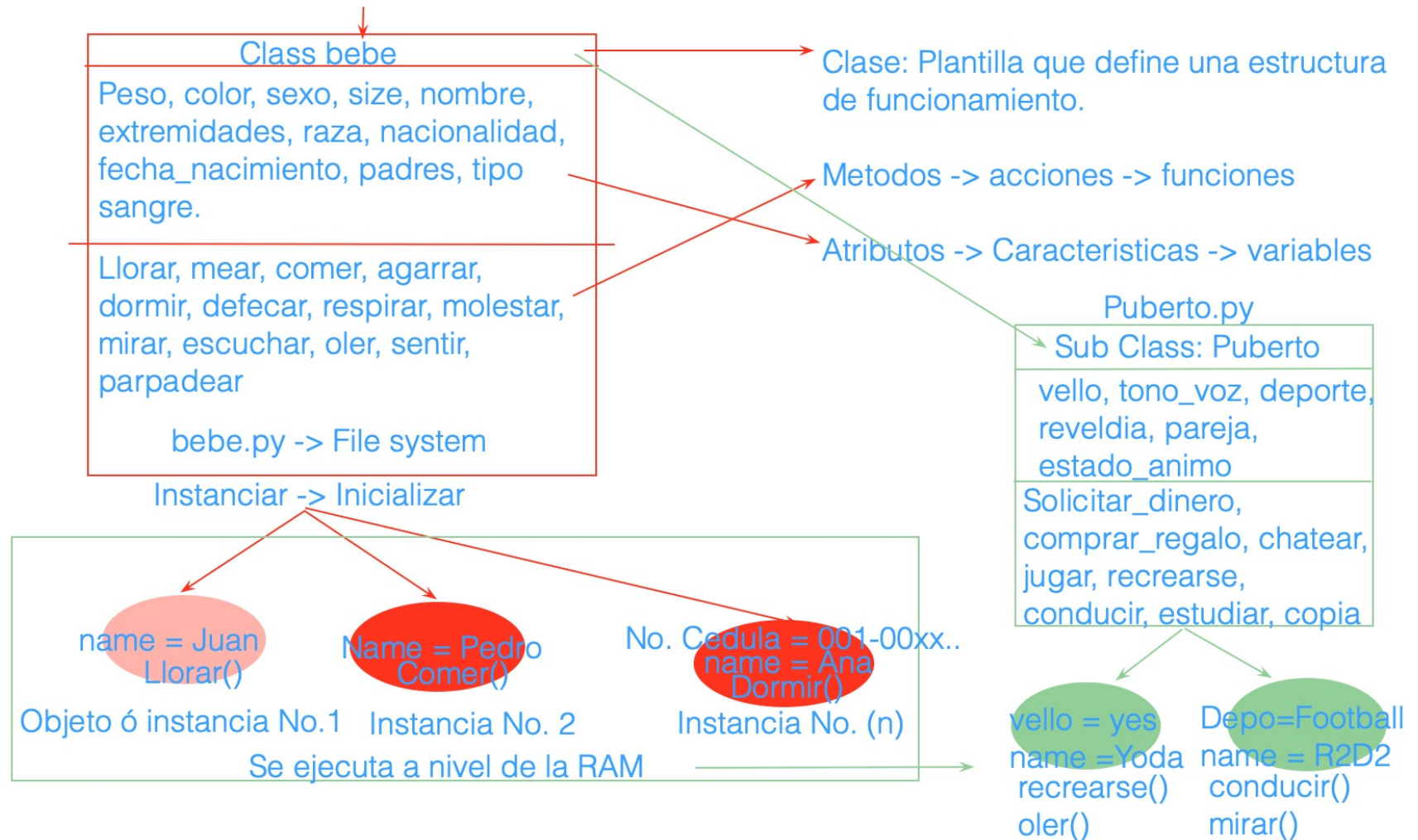
```
def questions(name, quest, favorite_color):  
    return name_prefix + name
```

- **When defining functions:**

- Place any necessary arguments that need to be passed into the function in parentheses.
- Insert a docstring at the very first line of the function's code, detailing the behavior of the function.
- Use the docstring to prescribe the function. Word it like a command, not a description.
- Use `return` to pass values out of the function when it is called, especially when you need to assign the call to a variable.
- Make sure the variables used in a function are within the proper scope.
- Avoid using global variables in functions when possible.

- **When calling functions:**

- Write the name of the function followed by open and closed parentheses.
- Provide the necessary arguments that the function will work with in the parentheses.
- Be mindful of which data types your arguments are.
- Be aware of which values, if any, are returned from the function.



```
class Knight:
    def __init__(self, name, quest, fav_color):
        self.name = name
        self.quest = quest
        self.fav_color = fav_color

    def display_name(self):
        print("welcome, Sir {}".format(self.name))
```



Formatting Class Names

Based on Python's style guide, you should always define variables in CapWords format.


- **Correct:** `class MyClass`
- **Incorrect:** `class my_class`

It's good practice to give your classes meaningful names in order to avoid ambiguity. You also cannot begin class names with a number. Python will produce a syntax error.

The `__init__()` function is required for initializing (constructing) instances of a class. It can take a number of arguments, the first of which is always `self`. This `self` argument refers to the instance that is being created by the class.


```
class Knight:
    def __init__(self, name, quest, favorite_color):
        self.name = name
        self.quest = quest
        self.favorite_color = favorite_color

    def display_name(self):
        print("Welcome, Sir {}".format(self.name))
```

 Operations

```
robin = Knight("Robin", "to seek the Holy Grail", "yellow")
robin.display_name()
```

Once the instance variables are initialized in `__init__()`, Python can use these in other methods within the class. For the `display_name` method, Python uses the `self.name` instance variable to, as the name suggests, display the knight's name.

Notice that you must first construct an instance of the class (`robin` in this case). Then, you can invoke a method on that instance. This results in:

```
Welcome, Sir Robin.
```


Instance Methods vs. Class Methods

instance. To create a method as a class method, add the `@classmethod` decorator on the line before the method definition. A *decorator* is simply an object that modifies the definition of a function, method, or class. Also, when you construct a class method, you'd typically use `cls` as its first argument, rather than `self`. For example, the following class `Knight` has a class method `population()`:

```
class Knight:
    count = 0

    def __init__(self):
        Knight.count += 1

    @classmethod
    def population(cls, knights):
        return "There are {} knights.".format(knights)
```

Assume you constructed four different instances of `Knight`. To take advantage of the `population()` class method and see how many instances you've constructed, you'd type:

```
>>> Knight.population(Knight.count)
"There are 4 knights."
```

Notice that the call is not attached to any instance—it is merely referencing the class (`Knight`) and the method (`population()`).

Unlike some other object-oriented programming languages, **classes in Python are dynamic**. In other words, you may add, change, or delete attributes of a class at any time. In the following code, the `robin` instance of class `Knight` adds attribute `age` and assigns it a value:

```
robin.age = 29
>>> Knight.age = 29
>>> print(robin.age)
29
```



Because `age` is now a valid attribute for the entire class, the instance `robin` can use it, as can other instances. You can also modify the attribute values of the instance or class by simply assigning it another value.

To delete an attribute from the instance:

```
del robin.age
```

Or, from the entire class:

```
del Knight.age
```

Python's dynamic class structure makes it easier to modify existing classes to fulfill a particular purpose, without you needing to actually edit that class directly. For example, say you import a module that provides some functionality to your program. You want to expand on this functionality to cater to your own program, but you don't want to change the module itself. **Altering attributes dynamically can help you achieve this.**

Because classes are dynamic, you may need to verify at some point if a class or instance still has a specific attribute. To do this, use the following syntax:

```
hasattr(instance_name, attribute_name)
```

```
if hasattr(robin, "age") is True:  
    print("Attribute exists.")
```

```
else:  
    print("Attribute does not exist.")
```

Elements inside of a class can be represented as dictionaries. These dictionaries list the attributes of a class or its instance. Assume that you still have the `robin` instance of the `Knight` class. You'll then add a few new attributes to the instance, as follows:

```
robin.age = 29
robin.crest = "bird"
robin.honorific = "the Brave"
```

`robin.__dict__`

≤ Operations

```
{'crest': 'bird', 'quest': 'to seek the Holy Grail', 'name': 'Robin',
'favorite_color': 'yellow', 'age': 29, 'honorific': 'the Brave'}
```

```
robin.__dict__["honorific"] = "the Not-So-Brave"
```



Note: If you add an attribute to an entire class dynamically, this attribute will not be in an instance's dictionary. You must assign the attribute to that particular instance.

A *property* is an attribute with getter, setter, and delete methods. These type of methods retrieve an attribute, modify an attribute, and delete an attribute, respectively. You'd typically use a property to streamline this behavior without needing to explicitly invoke each getter, setter, and delete method for an instance. For example, this is an example of the Knight class without a property:

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

    def get_name(self):
        return self.name

    def set_name(self, value):
        self.name = value

    def del_name(self):
        del self.name
```

Assume you constructed an instance of Knight called arthur:

```
arthur = Knight("Arthur")
```

Now, here's how you would normally invoke each method:

```
arthur.get_name
arthur.set_name = "King Arthur"
arthur.del_name
```

Properties – Python Styles I (property() function)

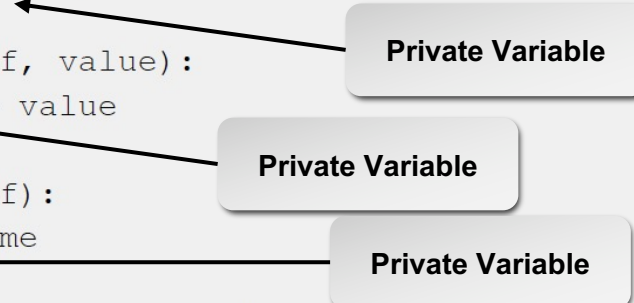
```
class Knight:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value

    def del_name(self):
        del self._name

    name = property(get_name, set_name, del_name)
```



Note the two major differences: the `property()` function at the bottom and the insertion of `_` before the name variable. Using this underscore before the variable makes the variable private. A private variable cannot be used outside of the class and is a requirement for using a property. Second, assigning the name variable to the property of each method allows you to bypass invoking these functions when you want to use them. Compare the following code to the three invocations above:

```
arthur.name
arthur.name = "King Arthur"
del arthur.name
```

As you can see, you no longer need to invoke a method to either get, set, or delete an attribute. You just need to reference the attribute and perform the operation like you would with any other variable. Python's `property()` function automatically invokes the relevant method in the class. This is particularly useful if you change method names; with a property, you won't have to change every invocation of these methods in your program. Using properties also makes for cleaner, easier-to-read code.

Properties - Python Styles II (@property)

Python has an object called a *decorator* that you can use to modify the definition of a function, method, or class. Rather than using the `property()` function as above, you can alternatively implement a property with the `@property` decorator. The following code is equivalent to the class definition above:

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @name.deleter
    def name(self):
        del self._name
```

Notice that the class reuses `name` for the method definitions. The `@property` decorator initiates the property, and each decorator below that is adding the proceeding method to the property. You can get, set, and delete an attribute just like before, without explicitly invoking the method:

```
arthur.name
arthur.name = "King Arthur"
del arthur.name
```


The basic syntax for defining a class that inherits from other classes is:

```
class SubClass(SuperClass1, SuperClass2, SuperClass3):
```

In inheritance, the term *superclass* refers to any class that is *being inherited*. Likewise, a *subclass* is a class that *does the inheriting*. As you can see, a subclass can inherit multiple superclasses in Python.



Note: The subclass/superclass relationship is also referred to as a child/parent relationship.

When your class inherits a superclass, it is able to use all of the members of that superclass. Considering the following two class definitions. The first is a superclass (Citizen), and the second is a subclass (Knight) that inherits the first:

```
class Citizen:
    def __init__(self, name, occupation, birthplace):
        self.name = name
        self.occupation = occupation
        self.birthplace = birthplace

    def display_info(self):
        print("{} the {}, is from {}".format(self.name, self.occupation,
self.birthplace))

class Knight(Citizen):
    def knight_quest(self):
        print("{} you must seek Camelot.".format(self.name))
```

The Knight subclass is inheriting Citizen because a knight is a type of citizen. This "is-a" relationship is a common use case for inheritance.

Now, construct an instance of Knight:

```
arthur = Knight("Arthur", "King", "Great Britain")
```

This instance is constructed with arguments that are defined in the superclass, despite using the subclass. Try and guess what the following code will do with the arthur instance:

```
arthur.display_info()
arthur.knight_quest()
```

Each method invoked is from a different class, yet both work as intended:

```
Arthur, the King, is from Great Britain.
Arthur, you must seek Camelot.
```

Checking Class Relationships

When inheritance gets complex, it can be helpful to know the direction of the subclass-superclass relationship. You can check this with the `issubclass(SubClass, SuperClass)` function. If the

first argument is indeed a subclass of the second argument, the result returns true. Using the classes above:

```
issubclass(Knight, Citizen)
```

This will return true because `Knight` is a subclass of `Citizen`.

Likewise, you can check if a certain object is an instance of a class or subclass using `isinstance(instance, Class)`. If the first argument is indeed an instance of the class or subclass in the second argument, it will return true:

```
isinstance(arthur, Citizen)
```

Special methods are methods that are reserved by Python and perform certain tasks within a class. You've already seen the special method `__init__()`, but there are more. All special methods have two leading and trailing underscores. The following table describes some special methods used by Python.

Special Method	Description
<code>__init__()</code>	Use to initialize an instance.
<code>__del__()</code>	Use to destroy an instance.
<code>__setattr__()</code>	Use to override the attributes of a class.
<code>__getattr__()</code>	Use to retrieve the attributes of a class.
<code>__delattr__()</code>	Use to delete the attributes of a class.
<code>__str__()</code>	Use to print returned values as neatly-formatted strings.
<code>__int__()</code>	Use to print returned values as integers.
<code>__float__()</code>	Use to print returned values as floats.



Note: This is not an exhaustive list. For a complete list of special methods, navigate to <https://docs.python.org/3/reference/datamodel.html>.

Operator Overloading

You can also use certain special methods to perform a process called *operator overloading*. This allows your class to define how it handles operators. Failing to overload operators may result in errors. For example, consider the following code:

```
class Addition:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "The answer is {}".format(self.a)

operator1 = Addition(3)
operator2 = Addition(5)
print(operator1 + operator2)
```

Logical
Operations

This code attempts to add the values of two instances together, but will produce an error. To actually perform the addition operation, you can use the `__add__()` special method to overload the operator:

```
class Addition:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "The answer is {}".format(self.a)

    def __add__(self, other):
        return Addition(self.a + other.a)

operator1 = Addition(3)
operator2 = Addition(5)
print(operator1 + operator2)
```

This outputs 8 because the `__add__()` method is defining how it handles the `+` operator when an instance uses it. Technically, you could change the `+` operator in the `__add__()` method to whatever you wanted. For example:

Operator Overloading

The following table lists various operator overloading special methods:

<i>Special Method</i>	<i>Equivalent Expression</i>
<code>__add__()</code>	<code>a + b</code>
<code>__sub__()</code>	<code>a - b</code>
<code>__mul__()</code>	<code>a * b</code>
<code>__truediv__()</code>	<code>a / b</code>
<code>__floordiv__()</code>	<code>a // b</code>
<code>__mod__()</code>	<code>a % b</code>
<code>__pow__()</code>	<code>a ** b</code>
<code>__and__()</code>	<code>a & b</code>
<code>__or__()</code>	<code>a b</code>
<code>__eq__()</code>	<code>a == b</code>
<code>__ne__()</code>	<code>a != b</code>
<code>__gt__()</code>	<code>a > b</code>
<code>__lt__()</code>	<code>a < b</code>
<code>__ge__()</code>	<code>a >= b</code>
<code>__le__()</code>	<code>a <= b</code>
<code>__contains__()</code>	<code>a in b</code> <code>a not in b</code>

al
ations

Like with functions, classes in Python can be thought of in terms of scope. Besides instance variables, Python also has class variables. A *class variable* can be shared by all instances of a class. They are defined outside of any methods within the class to differentiate them from instance variables. For example, say you want to use the `Citizen` class to keep a running count of all constructed instances of the class. You can use a class variable to ensure that all instances share a common count to increment:

```
class Citizen:
    citizen_count = 0

    def __init__(self, name, occupation, birthplace):
        self.name = name
        self.occupation = occupation
        self.birthplace = birthplace
        Citizen.citizen_count += 1
```

Using the correct code above, the following will result in a count of 4:

```
>>> arthur = Citizen("Arthur", "King", "Great Britain")
>>> lancelot = Citizen("Lancelot", "Knight", "Great Britain")
>>> bedevere = Citizen("Bedevere", "Knight", "Great Britain")
>>> galahad = Citizen("Galahad", "Knight", "Great Britain")
>>> print("There are {} citizens of the realm.".format(Citizen.citizen_count))
There are 4 citizens of the realm.
```

Be mindful of the scope of your classes' elements, just as you would when only working with functions.

Guidelines for Defining and Using Classes

- Use classes to streamline code that draws from similar characteristics.
- Think of classes as templates for creating new objects.
- Define a class with the `class` statement and append a colon (`:`) to the end of the line.
- Indent the code in the class.
- Construct an instance of a class and provide arguments in the format:
`instance = Class(args)`.
- Use the `__init__()` method to initialize code for each instance.
- Define functions inside of classes to create methods which an instance can use.
- Use a method with an instance in the following format: `instance.method()` .
- Consider that you can modify the attributes of a class at any time.
- Create dictionaries from instances and classes to work with the data they contain in a more structured format.

Guidelines for Defining and Using Classes (Cont.)

- **Use properties to avoid having to explicitly call methods from a class. Use either:**
 - The `property()` function.
 - The `@property` decorator.
- **Optimize your code and minimize the time you spend writing it by leveraging class inheritance.**
- **Use inheritance when working with an "is-a" relationship.**
- **Keep track of the relationships between superclasses and subclasses by using the `issubclass()` function.**
- **Take advantage of Python's built-in special methods.**
- **Use operator overloading to perform calculations on class attributes.**
- **Use class variables for sharing values across all instances of a class.**
- **Keep the scope of your variables in mind when using classes.**

```
import random
```

```
random_num = random.randint(1, 10)
```



```
import random

random_num = random.randint(1, 10)
user_guess = int(input("Guess a number between 1 and 10: "))

def result(random_num, user_guess):
    if user_guess == random_num:
        print("Good guess! {} was the correct number.".format(random_num))
    else:
        print("Sorry, the correct number was {}".format(random_num))

result(random_num, user_guess)
```



Note: Python modules are not in any special format—most of them are just Python code.

```
from random import randint  
  
random_num = randint(1, 10)
```



From ... Import

The second type of import is a *selective import*. In a selective import, you specify the exact objects that you need from a module, and nothing else. When you do this, only the objects you specify are available to you. The advantage of a selective import is that you don't need to continually reference the module itself when calling objects. So, instead of calling `random.randint()` every time, you'd be able to just write `randint()`. The syntax for a selective import is as follows:

```
from random import *  
  
random_num = randint(1, 10)
```

Logical

While they may seem to be the best of both worlds, there is a pitfall to using universal imports. If you define any objects of your own that have the same name as the objects from the imported module, it will cause ambiguity. Your code may end up failing to work as intended. This is a lot

Major Python Modules in the Standard Library

Modules Bundled with Python

One of the major strengths of Python is that it has a large number of modules already built into the language. All of these modules together make up Python's *standard library*. Many of the modules in this library are actually readily available to you as .py files in the Python directory. For example, you can find the `random` module in Python 3.4 by navigating your file manager to `/Python34/Lib/`.

Standard Module/Library	Description
<code>datetime</code>	Provides classes for working with time and dates.
<code>time</code>	Allows you to work with Unix time values.
<code>calendar</code>	Allows you to output calendars.
<code>math</code>	Provides advanced mathematical functions.
<code>random</code>	Allows you to generate pseudorandom numbers.
<code>re</code>	Allows you to perform regular expressions.
<code>csv</code>	Streamlines reading from and writing to CSV files.
<code>os</code>	Allows you to access OS-level functionality, including the file system.
<code>tkinter</code>	Provides tools for GUI programming.
<code>sys</code>	Provides access to resources used by the Python interpreter.



Note: Some modules, like `math`, are built into the interpreter and do not exist as discrete .py files.

<https://docs.python.org/3/library/>

Standard Module/Library	Description
<code>socket</code>	Provides a low-level networking interface.
<code>collections</code>	Provides additional data structures.
<code>json</code>	Provides encoding and decoding for JSON.
<code>shutil</code>	Allows you to perform high-level file operations like copying and moving.
<code>urllib</code>	Provides modules for processing web URLs.
<code>logging</code>	Provides event logging functionality.
<code>itertools</code>	Provides additional tools for iterative operations.
<code>functools</code>	Provides tools for working with higher-order functions.
<code>unittest</code>	Provides tools for creating and running tests on your code.
<code>argparse</code>	Provides tools for parsing command line arguments.

External Libraries and Modules

Even though Python comes with a sizeable standard library, there are numerous custom modules available that can extend the language even further. These custom modules are written by many different individuals and many different organizations, each one tailored to fill a certain need. There are various websites that compile lists of custom modules that may be useful to you, including

<https://wiki.python.org/moin/UsefulModules>

- Use modules to streamline your code and make the task of programming more efficient.
- Consult Python's standard library.
- Use a general import to leverage the full power of a module (`import module`).
- Use `from module import object` to perform a selective import.
- Use selective imports to only import the objects you need, and to avoid having to reference the module for each object call.
- Avoid using universal imports.
- Place import statements at the beginning of your source code.
- Research custom modules and libraries.

1. What examples can you think of that demonstrate when to use a Module versus just defining custom functions?
2. What kind of external modules will you search for to extend your program's capabilities?

