Module-4

Modules ***(Pg 145 – 154):*** Random numbers, the time module, the math module, creating your own modules, Namespaces, Scope and lookup rules, Attributes and the dot Operator, Three import statement variants.

Mutable versus immutable and aliasing ***(Pg 157 – 158)***

Object oriented programming ***(Pg 175 – 181) :*** Classes and Objects — The Basics, Attributes, Adding methods to our class, Instances as arguments and parameters, Converting an instance to a string, Instances as return values.

Chapters**: 8.1-8.8, 9.1, 11.1 – 11.8**

Modules (Pg 145– 154)

Random numbers, the time module, the math module, creating your own modules, Namespaces, Scope and lookup rules, Attributes and the dot Operator, Three import statement variants.

- A ***module*** is a ***file containing Python definitions and statements*** intended for ***use in other Python programs***. There are many Python modules that come with Python as part of the standard library.
- We have seen two of these already
  - the ***turtle module*** and
  - the ***string module***.

Turtle Module (Graphics in Python)

- The turtle module is a built-in Python library used to create graphics and drawings. It works like a pen (turtle) that moves on the screen and draws shapes.

Key Concepts

- **Turtle (pen)** → draws on screen
- **Screen (canvas)** → where drawing appears
- Movement is controlled using commands

Common Functions

• ***forward(x):*** **Move forward by x units** • ***backward(x):*** **Move backward** • ***left(angle):*** **Turn left** • ***right(angle):*** **Turn right**	• ***penup():*** **Lift pen (no drawing)** • ***pendown():*** **Start drawing** • ***color():*** **Set color** • ***circle(r):*** **Draw a circle**

Example 1: Draw a Square
import ***turtle***
t = turtle.Turtle()
 	
for i in range(4):
  t.forward(100)
  t.right(90)
turtle.done()
	Example 2: Draw a Circle
	import ***turtle***
	
	t = turtle.Turtle()
	t.circle(50)
	
	turtle.done()

Applications

- Teaching programming - basics
- Drawing shapes and patterns
- Simple animations

String Module (Working with Text)

- **What is a String?**
  - A string is a sequence of characters (text) enclosed in quotes.
  - Example: ***name = "Python"***
- **What is the String Module?**

  Python provides:
  - Built-in string functions (methods)
  - A separate module called string (for constants and utilities)

String Methods (Most Important)

- Method - Description
  - ***upper()*** ..... Convert to uppercase
  - ***lower()***..... Convert to lowercase
  - ***strip()***..... Remove spaces
  - ***replace()***..... Replace text
  - ***split()***..... Split into list
  - ***find()***..... Find position

Example 1: String Methods

```
text = "  hello python  "
print(text.strip())
# remove spaces
print(text.upper())
# HELLO PYTHON
print(text.replace("python", "world"))
```

Example 2: Split String

```
sentence = "Python is easy"

words = sentence.split()
print(words)
```

string Module (Constants)

You need to import it: ***import string***

Constant	*Meaning*
string.ascii_letters	a-z + A-Z
string.digits	0–9
string.punctuation	Special characters

Example 3: Using string module

```
import string

print(string.ascii_lowercase)
print(string.digits)
print(string.punctuation)
Random numbers
```

==========

Random numbers (Repeatability and Testing - Picking balls from bags, throwing dice, shuffling a pack of cards)

We often want to ***use random numbers in programs***; here are a few ***typical uses***:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin
- To shuffle a deck of playing cards randomly
- To allow/make an enemy spaceship appear at a random location and start shooting at the player
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam
- For encrypting banking sessions on the Internet.

Python provides a module random that helps with tasks like this. You can look it up using help, but here are the key things we'll do with it:

Python Code	Explanation
import random ***rng = random.Random()*** ***dice_throw = rng.randrange(1,7)*** ***print(dice_throw)*** ***delay_in_seconds = rng.random() * 5.0*** ***Output: 4***	# built-in Python module / a black box object used to generate random numbers # creates a random number generator object. "rng" is a variable that stores this generator. # Return an int, one of 1,2,3,4,5,6 # generates a random decimal number between 0.0 and 1.0

Complete Program Flow

1. Import the random module.
2. Create a random number generator.
3. Generate a random number between 1 and 6 (dice).
4. Print the dice value.
5. Generate a random decimal delay between 0 and 5 seconds.

- The ***randrange()*** method call generates an integer between its lower and upper argument, using the same semantics as range—so the lower bound is included, but the upper bound is excluded. All the values have an equal probability of occurring (i.e. the results are uniformly distributed). Like ***range***, ***randrange()*** can also take an optional step argument.
- Let's assume we needed a random odd number less than 100, we could say:

  random_odd = rng.randrange(1, 100, 2)
- Other methods can also generate other distributions e.g.
  - a bell-shaped
  - "normal" distribution might be more appropriate for estimating seasonal rainfall
  - the concentration of a compound in the body after taking a dose of medicine.
- The random method returns a floating point number in the interval [0.0, 1.0) — the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right".
- This example shows how to shuffle a list. (shuffle cannot work directly with a lazy promise, so notice that we had to convert the range object using the list type converter first.)

  cards = list(range(52)) ***# Generate ints [0 .. 51] representing a pack of cards.***
  rng.shuffle(cards) ***# Shuffle the pack***

Repeatability and Testing

- Random number generators are based on a ***deterministic algorithm*** — ***repeatable*** and ***predictable***. So they're called ***pseudo-random generators*** — they are not genuinely random. They start with a seed value. Each time you ask for another random number, you'll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated.
- For debugging and for writing unit tests, it is convenient to have repeatability — programs that do the same thing every time they are run. We can arrange this by forcing the random number generator to be initialized with a known seed every time.

  1 drng = random.Random(123) # Create generator with known starting state
- This alternative way of creating a random number generator gives an explicit seed value to the object. Without this argument, the system probably uses something based on the time. So grabbing some random numbers from "***drng"*** today will give you precisely the same random sequence as it will tomorrow!

Picking balls from bags, throwing dice, shuffling a pack of cards

- Here is an example to generate a list containing ***"n"*** random int's between a lower and an upper bound:

```
import random

def make_random_ints(num, lower_bound, upper_bound):
    rng = random.Random()  # Create a random number generator
    result = []
    for i in range(num):
        result.append(rng.randrange(lower_bound, upper_bound))
    return result
>>> make_random_ints(5, 1, 13)  # Pick 5 random month numbers
[8, 1, 8, 5, 6]
```

- Notice that we got a duplicate in the result. Often this is wanted, e.g. If we throw a die five times, we expect some duplicates.
- But what if you don't want duplicates? If you wanted 5 distinct months, then this algorithm is wrong. In this case a good algorithm is to generate the list of possibilities, shuffle it, and slice off the number of elements you want:

```
xs = list(range(1,13)) #Make list 1..12  (there are no duplicates)
rng = random.Random()  #Make a random number generator
rng.shuffle(xs)        #Shuffle the list
result = xs[:5]        #Take the first five elements
```

- In statistics courses, the first case — allowing duplicates —is usually described as pulling balls out of a bag with replacement—you put the drawn ball back in each time, so it can occur again. The latter case, with no duplicates, is usually described as pulling balls out of the bag without replacement. Once the ball is drawn, it doesn't go back to be drawn again. TV lotto games work like this.
- The second "shuffle and slice" algorithm would not be so great if you only wanted a few elements, but from a very large domain. Suppose I wanted five numbers between one and ten million, without duplicates. Generating a list of ten million items, shuffling it, and then slicing off the first five would be a performance disaster! So let us have another try:

```
import random
def make_random_ints_no_dups(num, lower_bound, upper_bound):
    result = []
    rng = random.Random()
    for i in range(num):
        while True:
            candidate = rng.randrange(lower_bound, upper_bound)
            if candidate not in result:
                break
        result.append(candidate)
    return result
xs = make_random_ints_no_dups(5, 1, 10000000)
print(xs)
```

- This agreeably produces 5 random numbers, without duplicates:
  [3344629,1735163,9433892,1081511,4923270]

- Even this function has its pitfalls. Can you spot what is going to happen in this case?
  xs = make_random_ints_no_dups(10,1,6)

==========

The "time" module

- As we start to work with more sophisticated algorithms and bigger programs, a natural concern is "is our code efficient?" One way to experiment is to ***time*** how long various operations take. The ***"time"*** module has a function called ***clock*** that is recommended for this purpose. Whenever clock is called, it returns a floating point number representing how many seconds have elapsed since your program started running.
- The way to use it is to call clock and assign the result to a variable, say t0, just before you start executing the code you want to measure. Then after execution, call clock again, (this time we'll save the result in variable t1). The difference t1-t0 is the time elapsed, and is a measure of how fast your program is running.
- Let's try a small example. Python has a built-in sum function that can sum the elements in a list. We can also write our own. How do we think they would compare for speed? We'll try to do the summation of a list [0, 1, 2 ...] in both cases, and compare the results:

```
import time
def do_my_sum(xs):
    sum = 0
    for v in xs:
        sum += v
    return sum

sz = 10000000  # Lets have 10 million elements in the list
testdata = range(sz)

t0 = time.clock()
my_result = do_my_sum(testdata)
t1 = time.clock()
print("my_result = {0} (time taken = {1:.4f} seconds)"
        .format(my_result, t1-t0))

t2 = time.clock()
their_result = sum(testdata)
t3 = time.clock()
print("their_result = {0} (time taken = {1:.4f} seconds)"
        .format(their_result, t3-t2))
```

On a reasonably modest laptop, we get these results:

```
my_sum    = 49999995000000 (time taken = 1.5567 seconds)
their_sum = 49999995000000 (time taken = 0.9897 seconds)
```

- So our function runs about 57% slower than the built-in one. Generating and summing up ten million elements in under a second is not too thin!

==========

The "math" module

- The math module contains the kinds of mathematical functions you'd typically find on your ***calculator (sin, cos, sqrt, asin, log, log10)*** and some mathematical constants like ***pi*** and ***e***:

```
>>> import math
>>> math.pi        # Constant pi
3.141592653589793
>>> math.e         # Constant natural log base
2.718281828459045
>>> math.sqrt(2.0) # Square root function
1.4142135623730951
>>> math.radians(90) # Convert 90 degrees to radians
1.5707963267948966
>>> math.sin(math.radians(90)) # Find sin of 90 degrees
1.0
>>> math.asin(1.0)  * 2  # Double the arcsin of 1.0 to get pi
3.141592653589793
```

- The **math module** in Python provides a collection of **mathematical functions** to perform operations such as trigonometry, logarithms, exponentiation, and more.
- The **math module** is essential for performing **efficient mathematical computations** in Python. It provides **built-in optimized functions** for arithmetic, trigonometry, logarithms, and more, making programs simpler and faster.

Description	Example
Basic Mathematical Functions	
ceil() and floor() math.ceil(x) → returns the smallest integer ≥ x math.floor(x) → returns the largest integer ≤ x	import math print(math.ceil(4.2)) # Output: 5 print(math.floor(4.8)) # Output: 4
fabs() and factorial() • math.fabs(x) → absolute value • math.factorial(n) → factorial of n	print(math.fabs(-10)) # Output: 10.0 print(math.factorial(5)) # Output: 120
Power and Logarithmic Functions	
pow() and sqrt() math.pow(x, y) → x^y math.sqrt(x) → square root	print(math.pow(2, 3)) # Output: 8.0 print(math.sqrt(25)) # Output: 5.0
log() and log10() • math.log(x) → natural logarithm (base e) • math.log10(x) → logarithm base 10	print(math.log(10)) # Output: 2.302... print(math.log10(100)) # Output: 2.0
Trigonometric Functions (All angles are in radians)	
sin(), cos(), tan()	print(math.sin(math.pi/2)) # Output: 1.0 print(math.cos(0)) # Output: 1.0 print(math.tan(math.pi/4)) # Output: 1.0

degrees() and radians()
• **Convert between degrees and radians**
	print(math.degrees(math.pi))
	# Output: 180.0
	print(math.radians(180))
	# Output: 3.1415...

Constants in Math Module	
• **math.pi** → 3.14159 • **math.e** → 2.71828	print(math.pi) print(math.e)
• **exp()** - Returns e^x • **gcd()** - Greatest Common Divisor	print(math.exp(2)) # Output: 7.389... print(math.gcd(12, 18)) # Output: 6

- Complete Example Program

```
import math

num = 16

print("Square root:", math.sqrt(num))
print("Power:", math.pow(2, 4))
print("Log:", math.log(num))
print("Ceil:", math.ceil(4.3))
print("Floor:", math.floor(4.7))
print("Factorial:", math.factorial(5))
print("Sin(90):", math.sin(math.pi/2))
print("GCD:", math.gcd(20, 8))
```

==========

Creating our own modules

- All we need to do to create our own modules is to save our script as a file with a ***.py*** extension. Suppose, for example, this script is saved as a file named ***seqtools.py.***
- We can now use our module, both in scripts we write, or in the interactive Python interpreter. To do so, we must first import the module.

seqtools.py:

```
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4,s)
'A sting!'
```

- We do not include the ***.py*** file extension when importing. Python expects the file names of Python modules to end in ***.py,*** so the file extension is not included in the import statement.
- The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

==========

Name spaces

- A ***name space is a collection of identifiers that belong to a module, or to a function***.
- Generally, we like a namespace to hold "related" things, e.g. all the math functions, or all the typical things we'd do with random numbers. Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.

```
# module1.py
question = "What is the meaning of Life"
answer = 42
```

```
# module2.py
question = "What is your quest?"
answer = "To seek the holy grail."
```

- We can now import both modules and access ***question*** and ***answer*** in each:

```
import module1
import module2
print(module1.question)
print(module2.question)
print(module1.answer)
print(module2.answer)
```

output

```
What is the meaning of Life
What is your quest?
42
To seek the holy grail.
```

- Functions also have their own namespaces:

```
def f():
    n = 7
    print("printing n inside of f:", n)
def g():
    n = 42
    print("printing n inside of g:", n)
n = 11
print("printing n before calling f:", n)
f()
print("printing n after calling f:", n)
g()
print("printing n after calling g:", n)
```

output:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

- The three ***n's*** here do not collide since they are each in a different namespace—they are three names for three different variables.
- Namespaces permit several programmers to work on the same project without having naming collisions.

How are namespaces, files and modules related?

Python has a convenient and simplifying one-to-one mapping, one module per file, giving rise to one namespace. Also, Python takes the module name from the file name, and this becomes the name of the namespace. `math.py` is a filename, the module is called `math`, and its namespace is `math`. So in Python the concepts are more or less interchangeable.

So a good idea is to try to keep the concepts distinct in your mind.

Files and directories organize *where* things are stored in our computer. On the other hand, namespaces and modules are a programming concept: they help us organize how we want to group related functions and attributes. They are not about "where" to store things, and should not have to coincide with the file and directory structures.

So in Python, if you rename the file `math.py`, its module name also changes, your `import` statements would need to change, and your code that refers to functions or attributes inside that namespace would also need to change.

==========

Scope and lookup rules

- ***The scope of an identifier is the region of program code in which the identifier can be accessed, or used.***
- There are three important scopes in Python:
  - ***Local scope*** refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.
  - ***Global scope*** refers to all the identifiers declared within the current module, or file.
  - ***Built-in scope*** refers to all the identifiers built into Python — those like range and min that can be used without having to import anything, and are (almost) always available.
- Python can help you by telling you what is in which scope. Use the ***functions locals, globals***, and ***dir*** to see for yourself!
- Python uses precedence rules: the same name could occur in more than one of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope.

- Let's start with a simple example:

```
def range(n):
    return 123*n
print(range(10))
```

- What gets printed? We've defined our own function called ***range***, so there is now a potential ambiguity. When we use ***range***, do we mean our own one, or the built-in one? Using the scope lookup rules determines this: our own ***range*** function, not the built-in one, is called, because our function ***range*** is in the global namespace, which takes precedence over the built-in names.
- So although names likes ***range*** and ***min*** are built-in, they can be "hidden" from your use if you choose to define your own variables or functions that reuse those names.

```
n = 10
m = 3
def f(n):
    m = 7
    return 2*n+m
print(f(5), n, m)
```

OUTPUT

17 10 3

- Now, a slightly more complex example:
- The reason is that the two variables ***"m"*** and ***"n"*** in lines 1 and 2 are outside the function in the ***global namespace***. Inside the function, new variables called ***"n"*** and ***"m"*** are created just for the duration of the execution of ***"f"***. These are created in the local namespace of function ***"f"***.
- Within the body of ***"f"***, the scope lookup rules determine that we use the local variables ***"m"*** and ***"n"***.
- By contrast, after we've returned from ***"f"***, the ***"n"*** and ***"m"*** arguments to the print function refer to the original variables on lines 1 and 2, and these have not been changed in any way by executing function ***"f"***.
- Notice too that the def puts name f into the global namespace here. So it can be called on line 6. What is the scope of the variable ***"n"*** on line 1? Its scope — the region in which it is visible — is lines 1, 2, 6. It is hidden from view in lines 3, 4, 5 because of the local variable ***"n"***.

==========

Attributes and the dot(.) Operator

- ***Variables defined inside a module*** are ***called attributes of the module***.
- We've seen that objects have attributes too: for example, most objects have a ***__doc__*** attribute, some functions have a ***__annotations__*** attribute.
- Attributes are accessed using the ***dot operator (.).***
- The question attribute of module1 and module2 is accessed using module1.question and module2.question.
- An **attribute** is a named value that belongs to an object.

Types of Attributes

- **Instance Attributes** – Belong to a specific object
- **Class Attributes** – Shared by all objects of a class

Instance Attribute Example

```
class Student:
    def __init__(self, name, age):
        self.name = name    # instance attribute
        self.age = age

s1 = Student("Ravi", 20)
print(s1.name)    # Access using dot operator
print(s1.age)
```

Class Attribute Example

```
class Student:
    college = "VTU"    # class attribute

s1 = Student()
s2 = Student()

print(s1.college)
print(s2.college)
```

Dot Operator (.)

- The dot operator is used to:
  - Access attributes
  - Call methods
  - Access module functions

Syntax

  - object.attribute
  - object.method()
  - module.function()

Examples of Dot Operator

a) Accessing Object Attributes

```
class Car:
    def __init__(self, brand):
        self.brand = brand

c = Car("Toyota")
print(c.brand)
```

b) Calling Methods

```
class Car:
    def start(self):
        print("Car started")
c = Car()
c.start()
```

c) Accessing Module Functions

```
import math
print(math.sqrt(16))
# dot operator used
```


Important Points

- Attributes are accessed using dot (.) notation
- Objects can have multiple attributes
- Methods are also accessed using the dot operator
- Modules use dot notation to access functions and constants

Advantages

- Easy and readable syntax
- Helps in organizing code using objects
- Supports object-oriented programming

Conclusion

- Attributes store data in objects, and the dot operator provides a simple and effective way to access attributes, methods, and module functions, making Python programs more structured and readable.

==========

Three import statement variants

- Here are three different ways to import names into the current namespace, and to use them:
  import math
  x = math.sqrt(10)
- Here just the single identifier math is added to the current namespace. If you want to access one of the functions in the module, you need to use the dot notation to get to it.
- Here is a different arrangement:
  from math import cos, sin, sqrt
  x = sqrt(10)
- The names are added directly to the current namespace, and can be used without qualification. The name ***math*** is not itself imported, so trying to use the qualified form ***math.sqrt*** would give an error.

- Then we have a convenient shorthand:

  from math import *

  x = sqrt(10)
- Of these three, the first method is generally preferred, even though it means a little more typing each time.
- Although, we can make things shorter by importing a module under a different name:

  >>> import math as m

  >>> m.pi

  3.141592653589793
- Finally, observe this case:

```
def area(radius):
    import math
    return math.pi * radius * radius
x = math.sqrt(10) # This gives an error
```

- Here we imported math, but we imported it into the local namespace of area. So the name is usable within the function body, but not in the enclosing script, because it is not in the global namespace.

==========

Glossary

attribute A variable defined inside a module (or class or instance – as we will see later). Module attributes are accessed by using the **dot operator** (`.`).

dot operator The dot operator (`.`) permits access to attributes and functions of a module (or attributes and methods of a class or instance – as we have seen elsewhere).

fully qualified name A name that is prefixed by some namespace identifier and the dot operator, or by an instance object, e.g. `math.sqrt` or `tess.forward(10)`.

import statement A statement which makes the objects contained in a module available for use within another module. There are two forms for the import statement. Using hypothetical modules named `mymod1` and `mymod2` each containing functions `f1` and `f2`, and variables `v1` and `v2`, examples of these two forms include:

```
import mymod1
from mymod2 import f1, f2, v1, v2
```

We say that the method, `upper` is invoked on the string, `s`. `s` is implicitely the first argument to `upper`.

module A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the `import` statement.

namespace A syntactic container providing a context for names so that the same name can reside in different namespaces without ambiguity. In Python, modules, classes, functions and methods all form namespaces.

naming collision A situation in which two or more names in a given namespace cannot be unambiguously resolved. Using `import string` instead of `from string import *` prevents naming collisions.

Standard library: A library is a collection of software used as tools in the development of other software. The stan dard library of a programming language is the set of such tools that are distributed with the core programming language. Python comes with an extensive standard library.

==========

Mutable Vs immutable and aliasing (Pg 157 – 158)

- Some datatypes in Python are mutable. This means their contents can be changed after they have been created. **Lists and dictionaries** are **good examples of mutable** datatypes.

```
>>> my_list = [2, 4, 5, 3, 6, 1]
>>> my_list[0] = 9
>>> my_list
[9, 4, 5, 3, 6, 1]
```

Tuples and strings are examples of immutable datatypes, their contents can not be changed after they have been created:

```
>>> my_tuple = (2, 5, 3, 1)
>>> my_tuple[0] = 9
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Mutability is usually useful, but it may lead to something called aliasing. In this case, two variables refer to the same object and mutating one will also change the other:

```
>>> list_one = [1, 2, 3, 4, 6]
>>> list_two = list_one
>>> list_two[-1] = 5
>>> list_one
[1, 2, 3, 4, 5]
```

This happens, because both list_one and list_two refer to the same memory address containing the actual list. You can check this using the built-in function `id`:

```
>>> list_one = [1, 2, 3, 4, 6]
>>> list_two = list_one
>>> id(list_one) == id(list_two)
True
```

You can escape this problem by making a copy of the list:

```
>>> list_one = [1, 2, 3, 4, 6]
>>> list_two = list_one[:]
>>> id(list_one) == id(list_two)
False
>>> list_two[-1] = 5
>>> list_two
[1, 2, 3, 4, 5]
>>> list_one
[1, 2, 3, 4, 6]
```

However, this will not work for nested lists because of the same reason. The module `copy` provides functions to solve this.

==========

Object oriented programming *(Pg 175 – 181)*

Classes and Objects — The Basics, Attributes, Adding methods to our class, Instances as arguments and parameters, Converting an instance to a string, Instances as return values.

==========

Classes and Objects - The Basics

- Python is an object-oriented programming language, which means that it provides features that support object oriented programming (OOP).
- Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1980s that it became the main programming paradigm used in the creation of new software. It was developed as away to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time.
- Up to now, most of the programs we have been writing use a procedural programming paradigm. In procedural programming the focus is on writing functions or procedures which operate on data.
- In ***object-oriented programming*** the ***focus*** is on the creation of objects which contain ***both data and functionality together***.
- We have seen turtle objects, string objects, and random number generators, to name a few places where we've already worked with objects.
- Usually, each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

User-defined compound data types

==========

Attributes

==========

Adding other methods to our class

- Creating a class like ***Point*** brings an exceptional amount of "organizational power" to our programs, and to our thinking. We can group together the sensible operations, and the kinds of data they apply to, and each instance of the class can have its own state.
- A ***method*** behaves like a function but it is invoked on a specific instance, e.g. ***tess.right(90).*** Like a data attribute, methods are accessed using dot notation.
- Let's add another method, ***distance_from_origin,*** to see better how methods work:

```
class Point:
    def __init__(self, x=0, y=0):
        """ Create a new point at x, y """
        self.x = x
        self.y = y

    def distance_from_origin(self):
        """ Compute my distance from the origin """
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

- Let's create a few point instances, look at their attributes, and call our new method on them:

```
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.distance_from_origin()
5.0
>>> q = Point(5, 12)
>>> q.x
5
>>> q.y
12
>>> q.distance_from_origin()
13.0
>>> r = Point()
>>> r.x
0
>>> r.y
0
>>> r.distance_from_origin()
0.0
```

- When defining a method, the first parameter refers to the instance being manipulated. As already noted, it is customary to name this parameter self.
- Notice that the caller of ***distance_from_origin*** does not explicitly supply an argument to match the self parameter—this is done for us, behind our back.

==========

Instances as arguments and parameters

- We can pass an object as an argument in the usual way. We've already seen this in some of the turtle examples, where we passed the turtle to some function like ***draw_bar*** in the chapter titled Conditionals, so that the function could control and use what ever turtle instance we passed to it.
- ***Our variable only holds a reference to an object***, so passing ***tess*** into a function creates an alias: both the caller and the called function now have a reference, but there is only one turtle!
- Here is a simple function involving our new ***Point*** objects:

```
def print_point(pt):
    print("({0}, {1})".format(pt.x, pt.y))
```

- ***print_point*** takes a point as an argument and formats the output in whichever way we choose. If we call ***print_point(p)*** with point ***"p"*** as defined previously, the output is ***(3, 4).***

==========

Converting an instance to a string

- When we're working with ***classes*** and ***objects***, a preferred alternative is to ***add a new method*** to the ***class***.
- A better approach is to ***have a method*** so that every ***instance can produce a string representation*** of ***itself***. Let's initially call it ***to_string***:

```
class Point:
    def to_string(self):
        return "({0}, {1})".format(self.x, self.y)
```

Now we can say:

```
>>> p = Point(3, 4)
>>> print(p.to_string())
(3, 4)
```

But don't we already have a `str` type converter that can turn our object into a string? Yes! And doesn't `print` automatically use this when printing things? Yes again! But these automatic mechanisms do not yet do exactly what we want:

```
>>> str(p)
'<__main__.Point object at 0x01F9AA10>'
>>> print(p)
'<__main__.Point object at 0x01F9AA10>'
```

Python has a clever trick up its sleeve to fix this. If we call our new method `__str__` instead of `to_string`, the Python interpreter will use our code whenever it needs to convert a `Point` to a string. Let's re-do this again, now:

```
class Point:
    def __str__(self):     # All we have done is renamed the method
        return "({0}, {1})".format(self.x, self.y)
```

and now things are looking great!

```
>>> str(p)
(3, 4)
>>> print(p)
(3, 4)
```

==========

Instances as return values

- ***Functions*** and ***methods*** can ***return instances***. For example, given two ***Point*** objects, find their midpoint. First we'll write this as a regular function:

```
def midpoint(p1, p2):
    """ Return the midpoint of points p1 and p2 """
    mx = (p1.x + p2.x)/2
    my = (p1.y + p2.y)/2
    return Point(mx, my)
```

The function creates and returns a new `Point` object:

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = midpoint(p, q)
>>> r
(4.0, 8.0)
```

- Now let us do this as a method instead. Suppose we have a point object, and wish to find the midpoint halfway between it and some other target point:

```
class Point:
    def halfway(self, target):
        """ Return the halfway point"""
        mx = (self.x + target.x)/2
        my = (self.y + target.y)/2
        return Point(mx, my)
```

This method is identical to the function, aside from some renaming.
It's usage might be like this:

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = p.halfway(q)
>>> r
(4.0, 8.0)
```

- While this example assigns each point to a variable, this need not be done. Just as function calls are composable, method calls and object instantiation are also composable, leading to this alternative that uses no variables:

  >>> print(Point(3, 4).halfway(Point(5, 12)))
  (4.0, 8.0)

========== END OF MODULE 4 ==========