BIOS6642

Classes

Fuyong Xing

Department of Biostatistics and Informatics

Colorado School of Public Health

University of Colorado Anschutz Medical Campus

Outline

- **Classes**
- Class examples
- Other remarks

Classes

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
- A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated.
- Significant object-oriented languages include Java, C++, C#, **Python**, PHP, JavaScript, Ruby, Perl, Object Pascal, Scala, etc.

https://en.wikipedia.org/wiki/Object-oriented_programming

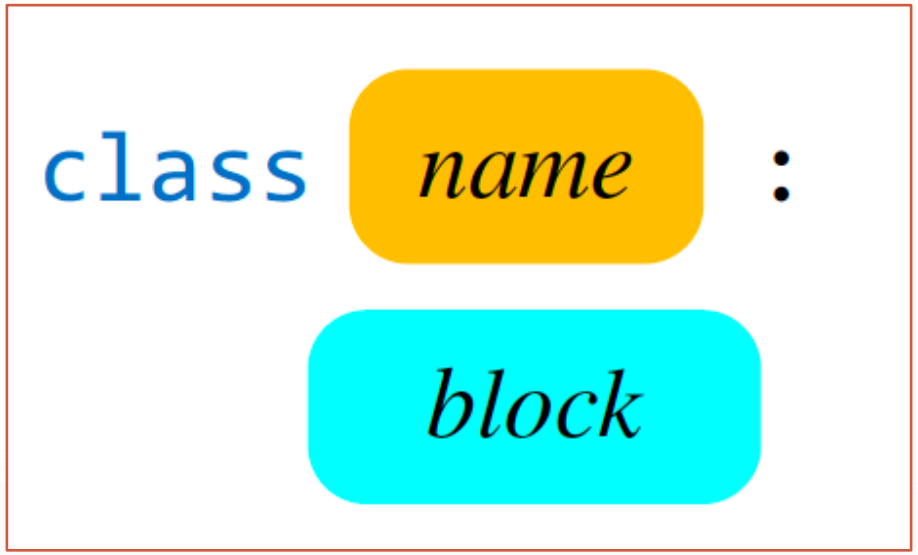
Classes

- In OOP, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).
- Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods for modifying its state.
- We can use class definitions to create custom types in Python.

https://en.wikipedia.org/wiki/Class_(computer_programming) and
https://docs.python.org/3/tutorial/classes.html#

Classes

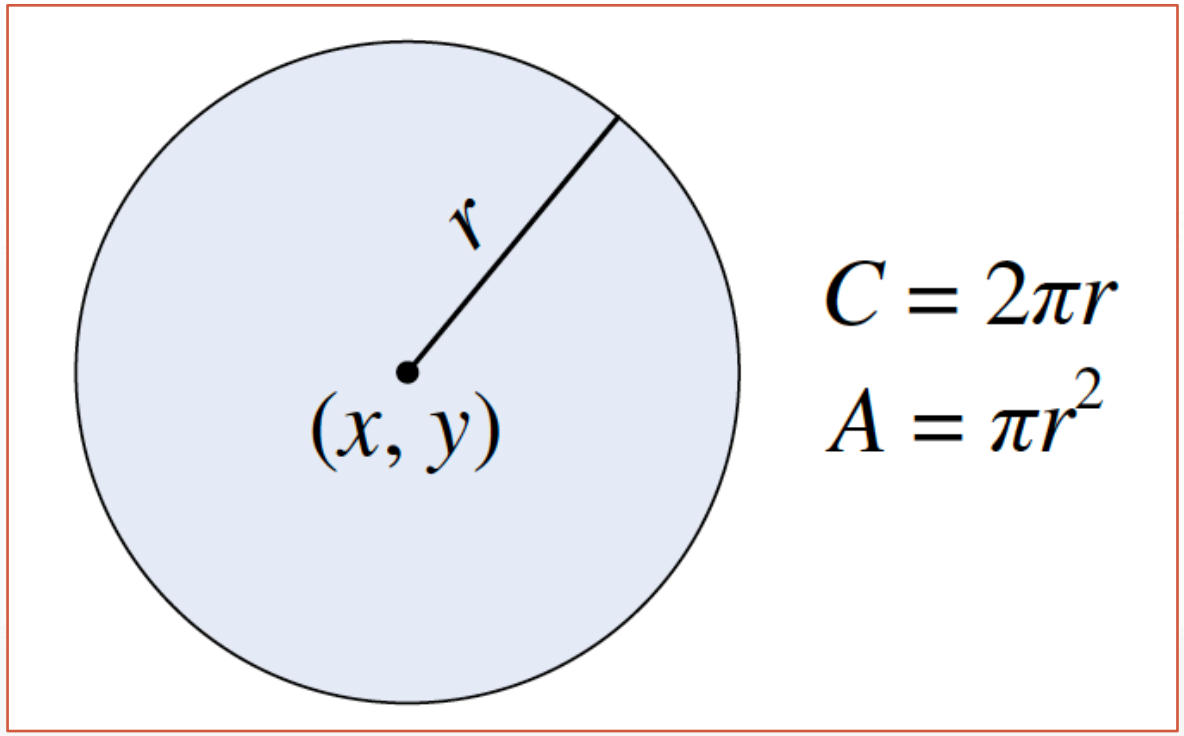
- Class definition



- A class definition begins with the reserved word *class* followed by the name of the class and a colon (:) at the end of the line.
- The body of the class is indented within a block of code. The code in this block often contains a series of method definitions.
- We can (and should) document classes and methods with docstrings.

Classes

- Example: create a Circle class.
- Mathematically, a circle can be defined with a center and a radius. Its circumference and area can be computed with the radius.



(x,y): center
r: radius
C: circumference
A: area

Classes

- Example: create a Circle class.
  - The Circle class specifies what circle objects can do and how clients (i.e., code outside of the Circle class needing the services that a object can provide) can interact with them.
  - Each object should maintain a center and a radius as the data.
  - The circumference and area can be implemented as methods.
  - Other methods can also be implemented to manipulate the data of each object.

Classes

- Example: create a Circle class.

A special method for class instantiation

```
class Circle:
    """  Represents a geometric circle object  """
    def __init__(self, center, radius):
        """  Initalize the center's center and radius  """
        # Disallow a negative radius
        if radius < 0:
            raise ValueError('Negative radius')
        self.center = center
        self.radius = radius

    def get_radius(self):
        """  Return the radius of the circle """
        return self.radius
```

```
    def get_center(self):
        """  Return the coordinatess of the center """
        return self.center

    def get_area(self):
        """  Compute and return the area of the circle  """
        from math import pi
        return pi*self.radius*self.radius

    def get_circumference(self):
        """  Compute and return the circumference of the circle  """
        from math import pi
        return 2*pi*self.radius

    def move(self, pt):
        """  Moves the enter of the circle to point pt """
        self.center = pt
```

```
def grow(self):
    """  Increases the radius of the circle """
    self.radius += 1

def shrink(self):
    """  Decreases the radius of the circle;
         does not affect a circle with radius zero  """
    if self.radius > 0:
       self.radius -= 1
```

Classes

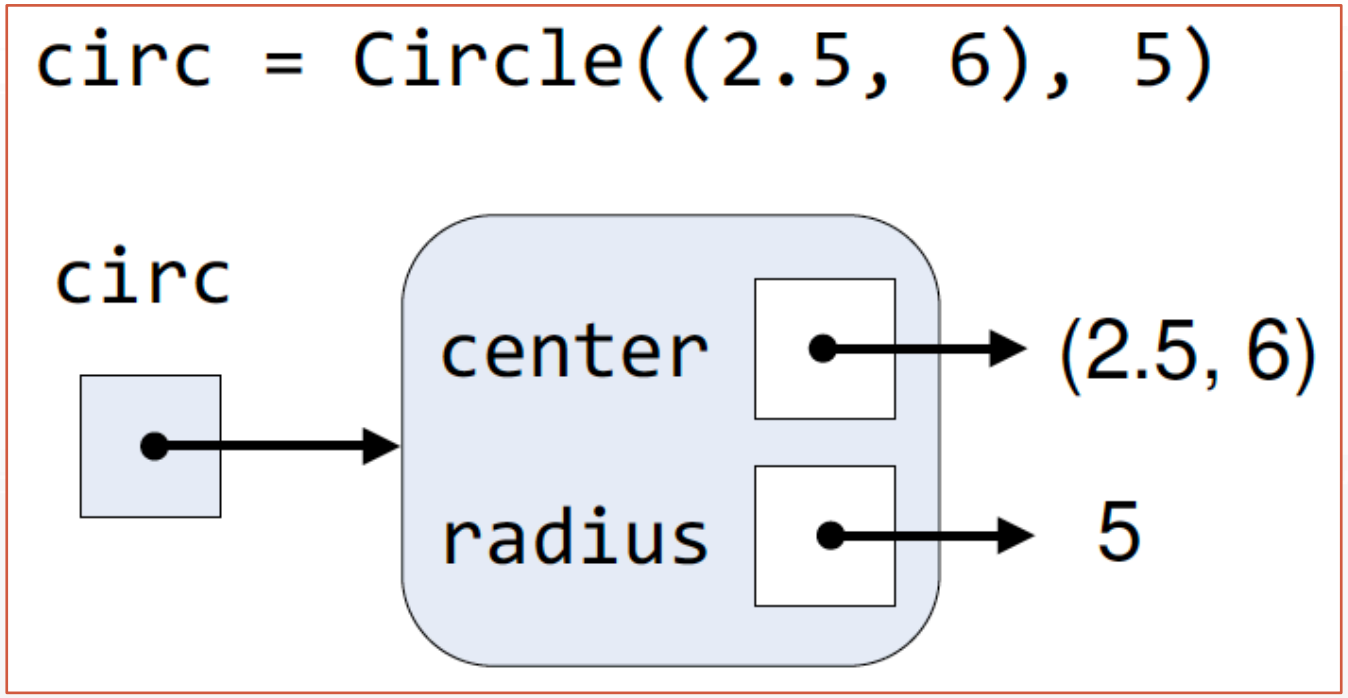
- Example: create a Circle class.
  - *__init__*: The special method for class instantiation in Python. The full name contains two underscore characters, followed by *init*, and then two more underscores. This method is automatically invoked when creating a new instance of the class (i.e., creating an object). It may or may not have parameters for class instantiation.
  - *get_radius*: This method simply returns the value of the radius instance variable.
  - *get_center*: This method simply returns the value of the center instance variable.
  - *get_area*: This method computes and returns an object's area.

Classes

- Example: create a Circle class.
  - *get_circumference:* This method computes and returns the circumference of an object.
  - *move*: This method repositions an object's center. The client must provide a tuple consisting of two numbers. This tuple represents the new coordinates of the object's center.
  - *grow*: This method increases an object's radius by one unit.
  - *shrink*: If an object's radius is greater than zero, this method decreases its radius by one unit. This method does not change the radius if the radius is zero before the call.

Classes

- The following statement creates a new object with a center at (2.5, 6) and radius 5.
- It implicitly invokes the *__init__* method for the instantiation, and the object is an instance of the class.
- It makes the variable, *circ*, refer to the object.



Classes

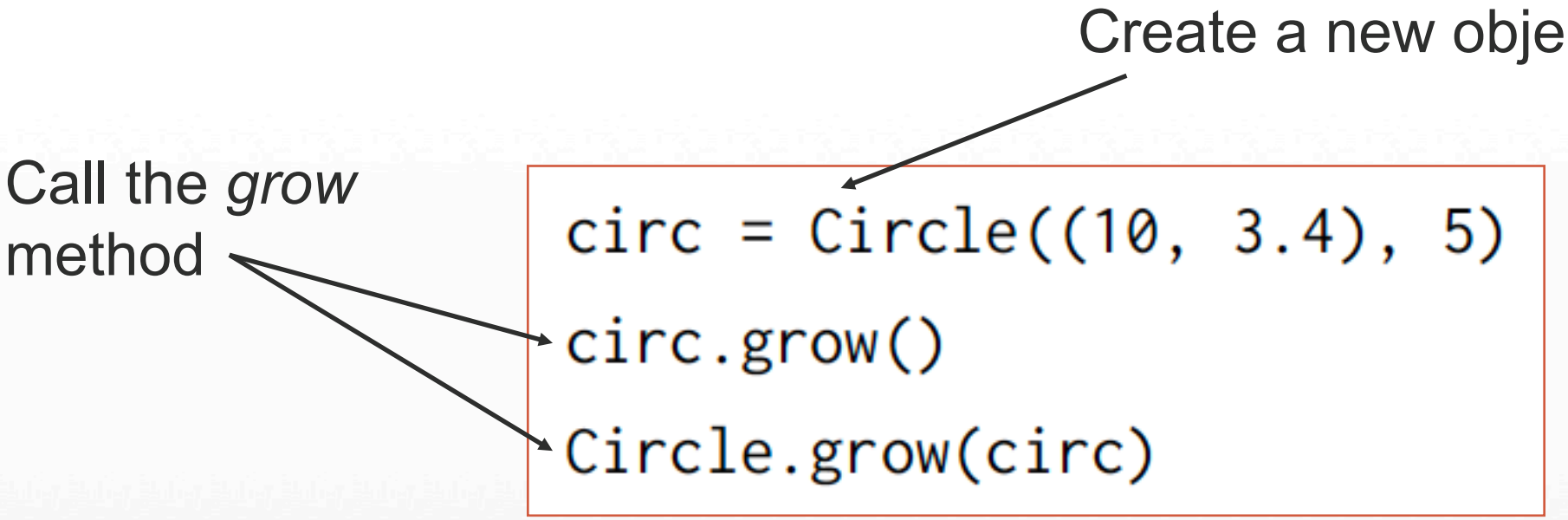
- Instance variables (or attributes)

```
class Circle:
    """  Represents a geometric circle object  """
    def __init__(self, center, radius):
        """  Initalize the center's center and radius  """
        # Disallow a negative radius
        if radius < 0:
            raise ValueError('Negative radius')
        self.center = center
        self.radius = radius
```

Instance variables are assigned values when the *__init__* method is invoked.

Classes

- In the Circle class definition, by convention, each method has *self* for its first parameter (although other valid identifiers can be used).
- When a method is called via the dot notation, the object is passed to the first parameter of the method.



Classes

- The instance variables can also be directly accessed with the dot notation.

```
>>> from circle import *
>>> c = Circle((0, 0), 4)
>>> c.get_radius()
4
>>> c.radius = 10
>>> c.get_radius()
10
```

Classes

- Python does not permit normal access with the dot notation to instance variables with names beginning with two underscores.

```
>>> class MyType:
...     def __init__(self):
...         self.__id = 2
...
>>> m = MyType()
>>> m.__id
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyType' object has no attribute '__id'
```

Classes

- Like regular functions, the methods within Python classes can accept default arguments, variable-length arguments, and keyword arguments.

```
class Circle:
    """  Represents a geometric circle object  """
    def __init__(self, center, radius=1):
        """  Initalize the center's center and radius  """
        # Disallow a negative radius
        if radius < 0:
            raise ValueError('Negative radius')
        self.center = center
        self.radius = radius


c1 = Circle((3, 17), 4)    # Radius specified to be 4 units
c2 = Circle((22, 8))       # Radius defaults to 1 unit
```

Outline

- Classes
- **Class examples**
- Other remarks

Class examples

- Build a simple database of customer bank accounts using classes
  - Every account has a unique identifier, the account number.
  - Each account's owner has a name.
  - Each account has a current balance.
  - The bank account management application could store the accounts in a list and move the contents of the list to a file for persistent storage.

```
class BankAccount:
    """  Models a bank account  """
    def __init__(self, number, name, balance):
        """  Initialize the instance variables of a bank account object.
             Disallows a negative initial balance.  """
        if balance < 0:
            raise ValueError('Negative initial balance')
        self.__account_number = number  # Account number
        self.__name = name              # Customer name
        self.__balance = balance        # Funds available in the account

    def id(self):
        """ Returns the account number of this bank account object.  """
        return self.__account_number

    def deposit(self, amount):
        """ Add funds to the account.  There is no limit
            to the size of the deposit.  """
        self.__balance += amount
```

```
def withdraw(self, amount):
    """ Remove funds from the account, if possible.
        Only completes the withdrawal successfully if
        there are enough funds in the account to
        fulfill the withdrawal.
        Return true if successful, false otherwise """
    result = False   # Unsuccessful by default
    if self.__balance - amount >= 0:
        self.__balance -= amount
        result = True   # Success
    return result

def __str__(self):
    """  Returns the string representation for this object """
    return '[{:>5} {:<10} {:>7} ]'.format(self.__account_number,
                                          self.__name, self.__balance)
```

A special method returning a string representation of an object

```
# ------------------------------------------------------------
# Global functions that use BankAccount objects

def open_database(filename, db):
    """ Read account information from a given file and store it
        in the given list.  """
    with open(filename) as lines:     # Open file to read
        for line in lines:
            line.strip()
            number, name, balance = line.split(",")
            db.append(BankAccount(int(number), name, int(balance)))


def print_database(db):
    """ Display the contents of the database """
    for acct in db:
        print(acct)  # Calls the __str__ method of acct
```

```
def get_account(db, account_number):
    """ Retrieve an account object with a given account number. """
    for acct in db:
        if acct.id() == account_number:
            return acct


def main():
    # Simple bank account "database"
    database = []

    try:
        # Open the database of accounts
        open_database('accountdata.text', database)
        print_database(database)
        print("--------------------")
        customer = get_account(database, 129)
        if customer:
            print("Account 129 before deposit of $100: ", end="")
```

```
            print(customer)
            customer.deposit(100)
            print("Account 129 after  deposit of $100: ", end="")
            print(customer)
            print("----------------------")
            print("Account 129 before withdraw of $500: ", end="")
            print(customer)
            customer.withdraw(500)
            print("Account 129 after  withdraw of $500: ", end="")
            print(customer)
            print("----------------------")
            print("Account 129 before withdraw of $80000: ", end="")
            print(customer)
    except Exception:
        print('Error in account database')
        raise


main()
```

Class examples

- Given a text file named *accountdata.text* containing the following data:

```
 324, 'Fred',     34523
 371, 'Ella',  1263210
 129, 'Zoe',    78934
 120, 'Owen',  247702
 412, 'Lily',  12000
 420, 'Bert',  10354
1038, 'George', 6733498
 966, 'Jan',   9923912
1210, 'Judy',     83497
1300, 'Sam',    50315
```

Class examples

Output

```
[  324 Fred          34523 ]
[  371 Ella        1263210 ]
[  129 Zoe          78934 ]
[  120 Owen        247702 ]
[  412 Lily        12000 ]
[  420 Bert        10354 ]
[ 1038 George       6733498 ]
[  966 Jan         9923912 ]
[ 1210 Judy           83497 ]
[ 1300 Sam          50315 ]
--------------------
Account 129 before deposit of $100: [  129 Zoe          78934 ]
Account 129 after  deposit of $100: [  129 Zoe          79034 ]
--------------------
Account 129 before withdraw of $500: [  129 Zoe          79034 ]
Account 129 after  withdraw of $500: [  129 Zoe          78534 ]
--------------------
Account 129 before withdraw of $80000: [  129 Zoe          78534 ]
Account 129 after  withdraw of $80000: [  129 Zoe          78534 ]
```

Class examples

- In the *BankAccount* class:
  - The constructor (__*init*__) instantiates objects, instances of class *BankAccount*, but it disallows the creation of an object with a negative account balance.
  - Clients may add funds via the *deposit* method.
  - The *withdraw* method prevents a client from withdrawing more money than the balance in the account. An overdraft attempt will not change the account's balance.
  - The special method, __*str*__, returns a string representation of an object. It can be invoked by the *print* or *str* functions to compute a nicely printable string representation of an object.

Class examples

- We note that the names of all instance variables begin with two underscores. It encourages the client to deposit or withdraw money using the *deposit* or *withdraw* method.

```
acct = BankAccount(31243, 'Joe', 1000)
acct.deposit(100)
acct.__balance -= 100    # Illegal
acct.withdraw(2000.00); # Method should disallow this operation
```

- Notice that the operations of depositing and withdrawing funds are the responsibility of the object itself, not the client code.

Class examples

- A Stopwatch class to measure elapsed time for a program's execution

```
from time import clock

class Stopwatch:
    """  Provides stopwatch objects that that programmers
         can use to time the execution time of portions of
         a program.  """
    def __init__(self):
        """  Makes a new stopwatch ready for timing.  """
        self.reset()

    def start(self):
        """  Starts the stopwatch, unless it is already running.
             This method does not affect any time that may have
             already accumulated on the stopwatch.  """
        if not self._running:
            self._start_time = clock() - self._elapsed
            self._running = True  # Clock now running
```

```
def stop(self):
    """  Stops the stopwatch, unless it is not running.
         Updates the accumulated elapsed time. """
    if self._running:
        self._elapsed = clock() - self._start_time
        self._running = False   # Clock stopped

def reset(self):
    """  Resets stopwatch to zero.  """
    self._start_time = self._elapsed = 0
    self._running = False

def elapsed(self):
    """  Reveals the stopwatch running time since it
         was last reset.  """
    if not self._running:
        return self._elapsed
    else:
        return clock() - self._start_time
```

Class examples

- In this Stopwatch class definition, four methods are available to clients: *start, stop, reset*, and *elapsed*.
- A client does not have to worry about the “messy” detail of the arithmetic to compute the elapsed time.
- A client using a Stopwatch object is much less likely to make a mistake because the details that make it work are hidden.

```
timer = Stopwatch()  # Declare a stopwatch object

timer.start()    # Start timing
#
#   Do something here that you wish to time
#
timer.stop()    # Stop the clock
print(timer.elapsed(), " seconds")
```

Class examples

- Use a Stopwatch object to compare the speed of two different prime number algorithms

```
""" Count the number of prime numbers less than
    two million and time how long it takes
    Compares the performance of two different
    algorithms. """

from stopwatch import Stopwatch
from math import sqrt

def count_primes(n):
    """
    Generates all the prime numbers from 2 to n - 1.
    n - 1 is the largest potential prime considered.
    """
    timer = Stopwatch()
```

```
timer.start()           # Record start time

count = 0
for val in range(2, n):
    root = round(sqrt(val)) + 1
    # Try all potential factors from 2 to the square root of n
    for trial_factor in range(2, root):
        if val % trial_factor == 0:  # Is it a factor?
            break                    # Found a factor
    else:
        count += 1                   # No factors found

timer.stop()    # Stop the clock
print("Count =", count, "Elapsed time:", timer.elapsed(), "seconds")
```

```
def sieve(n):
    """
    Generates all the prime numbers from 2 to n - 1.
    n - 1 is the largest potential prime considered.
    Algorithm originally developed by Eratosthenes.
    """

    timer = Stopwatch()
    timer.start()   # Record start time

    # Each position in the Boolean list indicates
    # if the number of that position is not prime:
    # false means "prime," and true means "composite."
    # Initially all numbers are prime until proven otherwise
    nonprimes = n * [False]  # Global list initialized to all False

    count = 0

    # First prime number is 2; 0 and 1 are not prime
    nonprimes[0] = nonprimes[1] = True
```

```
    # Start at the first prime number, 2.
    for i in range(2, n):
        # See if i is prime
        if not nonprimes[i]:
            count += 1
            # It is prime, so eliminate all of its
            # multiples that cannot be prime
            for j in range(2*i, n, i):
                nonprimes[j] = True
    # Print the elapsed time
    timer.stop()
    print("Count =", count, "Elapsed time:", timer.elapsed(), "seconds")


def main():
    count_primes(2000000)
    sieve(2000000)


main()
```

Outline

- Classes
- Class examples
- **Other remarks**

Other remarks

- Python allows programmers to add instance variables dynamically to individual objects.

Codes

```
class Widget:
    pass

w = Widget()
w.a = 16
w.t = 100
print(w.a, w.t)
```

Output

```
16 100
```

Other remarks

- Can check what instance variables associated with an object using a special attribute, __*dict*__.

```
>>> class Widget:
...     def __init__(self):
...             self.a = 10
...
>>> w = Widget()
>>> w.__dict__
{'a': 10}
>>> class Gadget:
...     pass
...
>>> g = Gadget()
>>> g.__dict__
{}
>>> g.x = 20
>>> g.y = 30
>>> g.__dict__
{'x': 20, 'y': 30}
```

Other remarks

- Can use built-in functions, *setattr* and *getattr*, to set and get the value of an instance variable.
- Suppose *g* is an instance of the class, *Gadget*.
  - The following two statements are equivalent:

```
g.x = 5

setattr(g, "x", 5)
```

  - The following two statements are also equivalent:

```
print(g.x)

print(getattr(g, "x"))
```

Other remarks

- Example

Codes

```
class Widget:
    def __init__(self):
        self.a = 5   #  Provide a preexisting attribute

w = Widget()
print(w.__dict__)
print("w.a =", getattr(w, "a"))
setattr(w, "a", 10)
print("w.a =", getattr(w, "a"))
field_name = input("Enter instance variable name: ")
setattr(w, field_name, 15)
print(getattr(w, field_name))
setattr(w, field_name, 20)
print(getattr(w, field_name))
print(w.__dict__)
```

Other remarks

- Example

Output

```
{'a': 5}
w.a = 5
w.a = 10
Enter instance variable name: my_field
15
20
{'a': 10, 'my_field': 20}
```

Other remarks

- Class variables: variables defined within a class but outside of all methods of that class. They are associated with the class instead of class instances.

Codes

```
class Widget:
    """ Models a manufactured item. """
    serial_number_source = 0 # Class variable
    def __init__(self):
        """ Make a widget with a unique serial number. """
        self.serial_number = Widget.serial_number_source
        Widget.serial_number_source += 1

    def get_serial_number(self):
        """ Return the widget's serial number. """
        return self.serial_number

if __name__ == '__main__':
    widget_list = []
    for i in range(10):
        widget_list.append(Widget())
    for w in widget_list:
        print(w.serial_number, end=' ')
    print()
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

Other remarks

- Operator overloading: A class can implement certain operations that are invoked by special syntax (such as arithmetic operations) by defining methods with special names.
- Operator overloading allows classes to define their own behavior of operators on programmer-defined types.
- For example, we can define a special method named __*add*__ for a class such that we can use the + operator on the instances of that class.
- More special methods can be found via https://docs.python.org/3/reference/datamodel.html#specialnames.

Other remarks

- Build a Time class to record the time of day

Codes

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
    def __init__(self, hour=0, minute=0, second=0):
        """Initializes a time object.

        hour: int
        minute: int
        second: int or float
        """
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        """Returns a string representation of the time."""
        return '{:02}:{:02}:{:02}'.format(self.hour, self.minute, self.second)
```

Codes

```
def print_time(self):
    """Prints a string representation of the time."""
    print(str(self))

def time_to_int(self):
    """Computes the number of seconds since midnight."""
    minutes = self.hour * 60 + self.minute
    seconds = minutes * 60 + self.second
    return seconds

def is_after(self, other):
    """Returns True if t1 is after t2; false otherwise."""
    return self.time_to_int() > other.time_to_int()

def __add__(self, other):
    """Adds two Time objects or a Time object and a number.

    other: Time object or number of seconds
    """
    if isinstance(other, Time):
        return self.add_time(other)
    else:
        return self.increment(other)

def __radd__(self, other):
    """Adds two Time objects or a Time object and a number."""
    return self.__add__(other)
```

Codes

```
    def add_time(self, other):
        """Adds two time objects."""
        assert self.is_valid() and other.is_valid()
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        """Returns a new Time that is the sum of this time and seconds."""
        seconds += self.time_to_int()
        return int_to_time(seconds)

    def is_valid(self):
        """Checks whether a Time object satisfies the invariants."""
        if self.hour < 0 or self.minute < 0 or self.second < 0:
            return False
        if self.minute >= 60 or self.second >= 60:
            return False
        return True


def int_to_time(seconds):
    """Makes a new Time object.

    seconds: int seconds since midnight.
    """
```

Codes

```
    minutes, second = divmod(seconds, 60)
    hour, minute = divmod(minutes, 60)
    time = Time(hour, minute, second)
    return time


def main():
    start = Time(9, 45, 0)
    start.print_time()

    end = start.increment(1337)
    #end = start.increment(1337, 460)
    end.print_time()

    print('Is end after start?')
    print(end.is_after(start))

    print('Using __str__')
    print(start, end)

    start = Time(9, 45)
    duration = Time(1, 35)
    print(start + duration)
    print(start + 1337)
    print(1337 + start)
```

```
if __name__ == '__main__':
    main()
```

Output

```
09:45:00
10:07:17
Is end after start?
True
Using __str__
09:45:00 10:07:17
11:20:00
10:07:17
10:07:17
```

Readings

- Think Python, chapters 15, 16, and 17

References

- *Think Python: How to Think Like a Computer Scientist*, 2nd edition, 2015, by Allen Downey
- *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones, 2013
- *Python for Data Analysis*, 2nd edition, by Wes McKinney, 2018
- *Fundamentals of Python Programming,* by Richard L. Halterman