Designing Data Types
Outline

1. APIs
2. Encapsulation
3. Immutability
4. Polymorphism
5. Overloading
6. Functions are Objects
7. Examples
8. Inheritance
9. Design by Contract
Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation.

By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we compose.

APIs should provide to clients just the methods they need and no others.
Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation.

By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we compose.

APIs should provide to clients just the methods they need and no others.
Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation

By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we compose

APIs should provide to clients just the methods they need and no others
Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation.

By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we compose.

APIs should provide to clients just the methods they need and no others.
Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation.

Encapsulation allows one implementation of an API to be substituted for another.

Encapsulation helps programmers ensure that their code operates as intended.

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named.

The API should be the only point of dependence between client and implementation — this is called modular programming.
Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation.

Encapsulation allows one implementation of an API to be substituted for another.

Encapsulation helps programmers ensure that their code operates as intended.

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named.

The API should be the only point of dependence between client and implementation — this is called modular programming.
Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation

Encapsulation allows one implementation of an API to be substituted for another

Encapsulation helps programmers ensure that their code operates as intended

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named

The API should be the only point of dependence between client and implementation — this is called modular programming
Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation.

Encapsulation allows one implementation of an API to be substituted for another.

Encapsulation helps programmers ensure that their code operates as intended.

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named.

The API should be the only point of dependence between client and implementation — this is called modular programming.
Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation.

Encapsulation allows one implementation of an API to be substituted for another.

Encapsulation helps programmers ensure that their code operates as intended.

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named.

The API should be the only point of dependence between client and implementation — this is called modular programming.
The process of separating clients from implementations by hiding information is known as encapsulation.

Encapsulation allows one implementation of an API to be substituted for another.

Encapsulation helps programmers ensure that their code operates as intended.

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named.

The API should be the only point of dependence between client and implementation — this is called modular programming.
An object from a data type is immutable if its data-type value cannot change once created.

The purpose of many data types (e.g., Stopwatch) is to encapsulate values that do not change, while for many other data types (e.g., Turtle), the very purpose of the abstraction is to encapsulate values as they change.

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types.

In Python, lists are mutable, whereas and strings and tuples are immutable.

The downside of immutability is that we must create a new object for every value, which is called defensive copy.

```python
class Vector:
    def __init__(self, a):
        self._coords = a[:]  # self._coords is a defensive copy of a
        self._n = len(a)
```
Immutability

An object from a data type is immutable if its data-type value cannot change once created.

The purpose of many data types (eg, Stopwatch) is to encapsulate values that do not change, while for many other data types (eg, Turtle), the very purpose of the abstraction is to encapsulate values as they change.

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types.

In Python, lists are mutable, whereas and strings and tuples are immutable.

The downside of immutability is that we must create a new object for every value, which is called defensive copy.

```python
class Vector:
    def __init__(self, a):
        self._coords = a[:] # self._coords is a defensive copy of a
        self._n = len(a)
```
An object from a data type is immutable if its data-type value cannot change once created.

The purpose of many data types (eg, Stopwatch) is to encapsulate values that do not change, while for many other data types (eg, Turtle), the very purpose of the abstraction is to encapsulate values as they change.

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types.

In Python, lists are mutable, whereas and strings and tuples are immutable.

The downside of immutability is that we must create a new object for every value, which is called defensive copy.

```python
class Vector:
    def __init__(self, a):
        self._coords = a[:]  # self._coords is a defensive copy of a
        self._n = len(a)
```
Immutability

An object from a data type is immutable if its data-type value cannot change once created.

The purpose of many data types (e.g., `Stopwatch`) is to encapsulate values that do not change, while for many other data types (e.g., `Turtle`), the very purpose of the abstraction is to encapsulate values as they change.

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types.

In Python, lists are mutable, whereas strings and tuples are immutable.

The downside of immutability is that we must create a new object for every value, which is called defensive copy.

```python
class Vector:
    def __init__(self, a):
        self._coords = a[:]  # self._coords is a defensive copy of a
        self._n = len(a)
```
Immutability

An object from a data type is immutable if its data-type value cannot change once created

The purpose of many data types (eg, Stopwatch) is to encapsulate values that do not change, while for many other data types (eg, Turtle), the very purpose of the abstraction is to encapsulate values as they change

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types

In Python, lists are mutable, whereas and strings and tuples are immutable

The downside of immutability is that we must create a new object for every value, which is called defensive copy

class Vector:
    def __init__(self, a):
        self._coords = a[:] # self._coords is a defensive copy of a
        self._n = len(a)
An object from a data type is immutable if its data-type value cannot change once created.

The purpose of many data types (e.g., `Stopwatch`) is to encapsulate values that do not change, while for many other data types (e.g., `Turtle`), the very purpose of the abstraction is to encapsulate values as they change.

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types.

In Python, lists are mutable, whereas both strings and tuples are immutable.

The downside of immutability is that we must create a new object for every value, which is called defensive copy.

```python
class Vector:
    def __init__(self, a):
        self._coords = a[:]  # self._coords is a defensive copy of a
        self._n = len(a)
```
A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function’s arguments

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information
Polymorphism

A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function’s arguments

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a TypeError at run time if an operation cannot be applied to an object because it is of an inappropriate type

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information
A method (or function) that can take arguments with different types is said to be polymorphic.

Duck typing is a programming style in which the language does not formally specify the requirements for a function’s arguments.

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type.

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type.

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information.
A method (or function) that can take arguments with different types is said to be polymorphic.

Duck typing is a programming style in which the language does not formally specify the requirements for a function’s arguments.

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type.

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type.

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information.
A method (or function) that can take arguments with different types is said to be polymorphic.

Duck typing is a programming style in which the language does not formally specify the requirements for a function’s arguments.

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type.

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type.

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information.
A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function’s arguments

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information
Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading.

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing.

We can also overload built-in functions, including absolute value, length, hashing, and type conversion.

Overloading operators and built-in functions makes user-defined types behave more like built-in types.

To perform an operation, Python internally converts the expression into a call on the corresponding special method.

To call a built-in function, Python internally calls the corresponding special method instead.

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code.
Overloading

Special methods for arithmetic operators

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>_add_(self, y)</td>
<td>sum of x and y</td>
</tr>
<tr>
<td>x - y</td>
<td>_sub_(self, y)</td>
<td>difference of x and y</td>
</tr>
<tr>
<td>x * y</td>
<td>_mul_(self, y)</td>
<td>product of x and y</td>
</tr>
<tr>
<td>x ** y</td>
<td>_pow_(self, y)</td>
<td>x to the power y</td>
</tr>
<tr>
<td>x / y</td>
<td>_div_(self, y)</td>
<td>quotient of x and y</td>
</tr>
<tr>
<td>x // y</td>
<td>_floordiv_(self, y)</td>
<td>floored quotient of x and y</td>
</tr>
<tr>
<td>x % y</td>
<td>_mod_(self, y)</td>
<td>remainder when dividing x by y</td>
</tr>
<tr>
<td>+x</td>
<td>_pos_(self)</td>
<td>x</td>
</tr>
<tr>
<td>-x</td>
<td>_neg_(self)</td>
<td>arithmetic negation of x</td>
</tr>
</tbody>
</table>

Special methods for comparison operators

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x == y</td>
<td>_eq_(self, y)</td>
<td>are x and y equal?</td>
</tr>
<tr>
<td>x != y</td>
<td>_ne_(self, y)</td>
<td>are x and y not equal?</td>
</tr>
<tr>
<td>x &lt; y</td>
<td>_lt_(self, y)</td>
<td>is x less than y?</td>
</tr>
<tr>
<td>x &lt;= y</td>
<td>_le_(self, y)</td>
<td>is x less than or equal to y?</td>
</tr>
<tr>
<td>x &gt; y</td>
<td>_gt_(self, y)</td>
<td>is x greater than y?</td>
</tr>
<tr>
<td>x &gt;= y</td>
<td>_ge_(self, y)</td>
<td>is x greater than or equal to y?</td>
</tr>
</tbody>
</table>
## Overloading

### Special methods for arithmetic operators

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td><strong>add</strong>(self, y)</td>
<td>sum of x and y</td>
</tr>
<tr>
<td>x - y</td>
<td><strong>sub</strong>(self, y)</td>
<td>difference of x and y</td>
</tr>
<tr>
<td>x * y</td>
<td><strong>mul</strong>(self, y)</td>
<td>product of x and y</td>
</tr>
<tr>
<td>x ** y</td>
<td><strong>pow</strong>(self, y)</td>
<td>x to the power y</td>
</tr>
<tr>
<td>x / y</td>
<td><strong>div</strong>(self, y)</td>
<td>quotient of x and y</td>
</tr>
<tr>
<td>x // y</td>
<td><strong>floordiv</strong>(self, y)</td>
<td>floored quotient of x and y</td>
</tr>
<tr>
<td>x % y</td>
<td><strong>mod</strong>(self, y)</td>
<td>remainder when dividing x by y</td>
</tr>
<tr>
<td>+x</td>
<td><strong>pos</strong>(self)</td>
<td></td>
</tr>
<tr>
<td>-x</td>
<td><strong>neg</strong>(self)</td>
<td>arithmetic negation of x</td>
</tr>
</tbody>
</table>

### Special methods for comparison operators

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x == y</td>
<td><strong>eq</strong>(self, y)</td>
<td>are x and y equal?</td>
</tr>
<tr>
<td>x != y</td>
<td><strong>ne</strong>(self, y)</td>
<td>are x and y not equal?</td>
</tr>
<tr>
<td>x &lt; y</td>
<td><strong>lt</strong>(self, y)</td>
<td>is x less than y?</td>
</tr>
<tr>
<td>x &lt;= y</td>
<td><strong>le</strong>(self, y)</td>
<td>is x less than or equal to y?</td>
</tr>
<tr>
<td>x &gt; y</td>
<td><strong>gt</strong>(self, y)</td>
<td>is x greater than y?</td>
</tr>
<tr>
<td>x &gt;= y</td>
<td><strong>ge</strong>(self, y)</td>
<td>is x greater than or equal to y?</td>
</tr>
</tbody>
</table>
### Overloading

#### Special methods for arithmetic operators

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td><strong>add</strong>(self, y)</td>
<td>sum of $x$ and $y$</td>
</tr>
<tr>
<td>x - y</td>
<td><strong>sub</strong>(self, y)</td>
<td>difference of $x$ and $y$</td>
</tr>
<tr>
<td>x * y</td>
<td><strong>mul</strong>(self, y)</td>
<td>product of $x$ and $y$</td>
</tr>
<tr>
<td>x ** y</td>
<td><strong>pow</strong>(self, y)</td>
<td>$x$ to the power $y$</td>
</tr>
<tr>
<td>x / y</td>
<td><strong>div</strong>(self, y)</td>
<td>quotient of $x$ and $y$</td>
</tr>
<tr>
<td>x // y</td>
<td><strong>floordiv</strong>(self, y)</td>
<td>floored quotient of $x$ and $y$</td>
</tr>
<tr>
<td>x % y</td>
<td><strong>mod</strong>(self, y)</td>
<td>remainder when dividing $x$ by $y$</td>
</tr>
<tr>
<td>+x</td>
<td><strong>pos</strong>(self)</td>
<td>arithmetic negation of $x$</td>
</tr>
<tr>
<td>-x</td>
<td><strong>neg</strong>(self)</td>
<td></td>
</tr>
</tbody>
</table>

#### Special methods for comparison operators

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x == y</td>
<td><strong>eq</strong>(self, y)</td>
<td>are $x$ and $y$ equal?</td>
</tr>
<tr>
<td>x != y</td>
<td><strong>ne</strong>(self, y)</td>
<td>are $x$ and $y$ not equal?</td>
</tr>
<tr>
<td>x &lt; y</td>
<td><strong>lt</strong>(self, y)</td>
<td>is $x$ less than $y$?</td>
</tr>
<tr>
<td>x &lt;= y</td>
<td><strong>le</strong>(self, y)</td>
<td>is $x$ less than or equal to $y$?</td>
</tr>
<tr>
<td>x &gt; y</td>
<td><strong>gt</strong>(self, y)</td>
<td>is $x$ greater than $y$?</td>
</tr>
<tr>
<td>x &gt;= y</td>
<td><strong>ge</strong>(self, y)</td>
<td>is $x$ greater than or equal to $y$?</td>
</tr>
</tbody>
</table>
## Special methods for built-in functions

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(x)</td>
<td><strong>len</strong>(self)</td>
<td>length of x</td>
</tr>
<tr>
<td>float(x)</td>
<td><strong>float</strong>(self)</td>
<td>float equivalent of x</td>
</tr>
<tr>
<td>int(x)</td>
<td><strong>int</strong>(self)</td>
<td>integer equivalent of x</td>
</tr>
<tr>
<td>str(x)</td>
<td><strong>str</strong>(self)</td>
<td>string representation of x</td>
</tr>
<tr>
<td>abs(x)</td>
<td><strong>abs</strong>(self)</td>
<td>absolute value of x</td>
</tr>
<tr>
<td>hash(x)</td>
<td><strong>hash</strong>(self)</td>
<td>integer hash code for x</td>
</tr>
<tr>
<td>iter(x)</td>
<td><strong>iter</strong>(self)</td>
<td>iterator for x</td>
</tr>
</tbody>
</table>
### Special methods for built-in functions

<table>
<thead>
<tr>
<th>client operation</th>
<th>special method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(x)</code></td>
<td><code>__len__(self)</code></td>
<td>length of $x$</td>
</tr>
<tr>
<td><code>float(x)</code></td>
<td><code>__float__(self)</code></td>
<td>float equivalent of $x$</td>
</tr>
<tr>
<td><code>int(x)</code></td>
<td><code>__int__(self)</code></td>
<td>integer equivalent of $x$</td>
</tr>
<tr>
<td><code>str(x)</code></td>
<td><code>__str__(self)</code></td>
<td>string representation of $x$</td>
</tr>
<tr>
<td><code>abs(x)</code></td>
<td><code>__abs__(self)</code></td>
<td>absolute value of $x$</td>
</tr>
<tr>
<td><code>hash(x)</code></td>
<td><code>__hash__(self)</code></td>
<td>integer hash code for $x$</td>
</tr>
<tr>
<td><code>iter(x)</code></td>
<td><code>__iter__(self)</code></td>
<td>iterator for $x$</td>
</tr>
</tbody>
</table>
Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results.

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing.

For example, the following function evaluates the Riemann integral (i.e., the area under the curve) of a real-valued function $f(\cdot)$ in the interval $(a, b)$, using the rectangle rule with $n$ rectangles:

```python
def integrate(f, a, b, n = 1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```

The following statement uses the above function to compute the area under the curve $f(x) = x^2$ in the interval $(0, 1)$:

```python
area = integrate(lambda x : x * x, 0, 1)
```
In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results.

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing.

For example, the following function evaluates the Riemann integral (i.e., the area under the curve) of a real-valued function $f()$ in the interval $(a, b)$, using the rectangle rule with $n$ rectangles:

```python
def integrate(f, a, b, n=1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```

The following statement uses the above function to compute the area under the curve $f(x) = x^2$ in the interval $(0, 1)$:

```python
area = integrate(lambda x: x * x, 0, 1)
```
Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing

For example, the following function evaluates the Riemann integral (i.e., the area under the curve) of a real-valued function \( f() \) in the interval \((a, b)\), using the rectangle rule with \( n \) rectangles

```python
def integrate(f, a, b, n = 1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```

The following statement uses the above function to compute the area under the curve \( f(x) = x^2 \) in the interval \((0, 1)\)

```python
area = integrate(lambda x : x * x, 0, 1)
```
Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results.

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing.

For example, the following function evaluates the Riemann integral (ie, the area under the curve) of a real-valued function \( f() \) in the interval \((a, b)\), using the rectangle rule with \( n \) rectangles:

```python
def integrate(f, a, b, n=1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```

The following statement uses the above function to compute the area under the curve \( f(x) = x^2 \) in the interval \((0, 1)\):

```python
area = integrate(lambda x : x * x, 0, 1)
```
Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results.

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing.

For example, the following function evaluates the Riemann integral (i.e., the area under the curve) of a real-valued function $f()$ in the interval $(a, b)$, using the rectangle rule with $n$ rectangles.

```python
def integrate(f, a, b, n=1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```

The following statement uses the above function to compute the area under the curve $f(x) = x^2$ in the interval $(0, 1)$.

```python
area = integrate(lambda x: x * x, 0, 1)
```
A data type `Complex` for complex numbers

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex(x, y)</td>
<td>a new complex object <code>c</code> with value (x + yi)</td>
</tr>
<tr>
<td><code>c.re()</code></td>
<td>real part of <code>c</code></td>
</tr>
<tr>
<td><code>c.im()</code></td>
<td>imaginary part of <code>c</code></td>
</tr>
<tr>
<td><code>c.conjugate()</code></td>
<td>conjugate of <code>c</code></td>
</tr>
<tr>
<td><code>c + d</code></td>
<td>sum of <code>c</code> and <code>d</code></td>
</tr>
<tr>
<td><code>c * d</code></td>
<td>product of <code>c</code> and <code>d</code></td>
</tr>
<tr>
<td><code>abs(c)</code></td>
<td>magnitude of <code>c</code></td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of <code>c</code></td>
</tr>
</tbody>
</table>

A complex number \(z\) in the polar form is expressed as
\[
z = re^{i\theta}
\]

Polar to cartesian: \(x = r\cos \theta\) and \(y = r\sin \theta\)

Cartesian to polar: \(r = \sqrt{x^2 + y^2}\) and \(\theta = \arctan(y/x)\)

If \(z_1 = r_1e^{i\theta_1}\) and \(z_2 = r_2e^{i\theta_2}\), then \(z_1z_2 = r_1r_2e^{i(\theta_1 + \theta_2)}\)
Examples

A data type `Complex` for complex numbers

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Complex(x, y)</code></td>
<td>a new complex object <code>c</code> with value <code>x + yi</code></td>
</tr>
<tr>
<td><code>c.re()</code></td>
<td>real part of <code>c</code></td>
</tr>
<tr>
<td><code>c.im()</code></td>
<td>imaginary part of <code>c</code></td>
</tr>
<tr>
<td><code>c.conjugate()</code></td>
<td>conjugate of <code>c</code></td>
</tr>
<tr>
<td><code>c + d</code></td>
<td>sum of <code>c</code> and <code>d</code></td>
</tr>
<tr>
<td><code>c * d</code></td>
<td>product of <code>c</code> and <code>d</code></td>
</tr>
<tr>
<td><code>abs(c)</code></td>
<td>magnitude of <code>c</code></td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of <code>c</code></td>
</tr>
</tbody>
</table>

A complex number `z` in the polar form is expressed as

\[ z = r e^{i \theta} \]

Polar to cartesian: \( x = r \cos \theta \) and \( y = r \sin \theta \)

Cartesian to polar: \( r = \sqrt{x^2 + y^2} \) and \( \theta = \arctan(y/x) \)

If \( z_1 = r_1 e^{i \theta_1} \) and \( z_2 = r_2 e^{i \theta_2} \), then \( z_1 z_2 = r_1 r_2 e^{i(\theta_1 + \theta_2)} \)
Examples

A data type `Complex` for complex numbers

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex(x, y)</td>
<td>a new complex object ( c ) with value ( x + yi )</td>
</tr>
<tr>
<td>c.re()</td>
<td>real part of ( c )</td>
</tr>
<tr>
<td>c.im()</td>
<td>imaginary part of ( c )</td>
</tr>
<tr>
<td>c.conjugate()</td>
<td>conjugate of ( c )</td>
</tr>
<tr>
<td>c + d</td>
<td>sum of ( c ) and ( d )</td>
</tr>
<tr>
<td>c * d</td>
<td>product of ( c ) and ( d )</td>
</tr>
<tr>
<td>abs(c)</td>
<td>magnitude of ( c )</td>
</tr>
<tr>
<td>str(c)</td>
<td>string representation of ( c )</td>
</tr>
</tbody>
</table>

A complex number \( z \) in the polar form is expressed as

\[
z = re^{i\theta}
\]

Polar to cartesian: \( x = r \cos \theta \) and \( y = r \sin \theta \)

Cartesian to polar: \( r = \sqrt{x^2 + y^2} \) and \( \theta = \arctan(y/x) \)

If \( z_1 = r_1e^{i\theta_1} \) and \( z_2 = r_2e^{i\theta_2} \), then \( z_1z_2 = r_1r_2e^{i(\theta_1+\theta_2)} \)
import math
import stdio

class Complex:
    def __init__(self, re=0.0, im=0.0):
        self._r = math.sqrt(re * re + im * im)
        self._theta = math.atan2(im, re)

    def re(self):
        return self._r * math.cos(self._theta)

    def im(self):
        return self._r * math.sin(self._theta)

    def conjugate(self):
        return Complex(self.re(), -self.im())

    def __add__(self, other):
        re = self.re() + other.re()
        im = self.im() + other.im()
        return Complex(re, im)

    def __mul__(self, other):
        c = Complex(0, 0)
        c._r = self._r * other._r
        c._theta = self._theta + other._theta
        return c
Examples

complexpolar.py: Complex data type redux.

```python
import math
import stdio

class Complex:
    def __init__(self, re=0.0, im=0.0):
        self._r = math.sqrt(re * re + im * im)
        self._theta = math.atan2(im, re)

    def re(self):
        return self._r * math.cos(self._theta)

    def im(self):
        return self._r * math.sin(self._theta)

    def conjugate(self):
        return Complex(self.re(), -self.im())

    def __add__(self, other):
        re = self.re() + other.re()
        im = self.im() + other.im()
        return Complex(re, im)

    def __mul__(self, other):
        c = Complex(0, 0)
        c._r = self._r * other._r
        c._theta = self._theta + other._theta
        return c
```
def __abs__(self):
    return self._r

def __str__(self):
    return str(self.re()) + ' + ' + str(self.im()) + 'i'

def _main():
    z0 = Complex(1.0, 1.0)
    z = z0
    z = z * z + z0
    z = z * z + z0
    stdio.writeln(z)

if __name__ == '__main__':
    _main()

$ python complexpolar.py
-7.0 + 7.0i
Examples

```python
def __abs__(self):
    return self._r

def __str__(self):
    return str(self.re()) + ' + ' + str(self.im()) + 'i'

def _main():
    z0 = Complex(1.0, 1.0)
    z = z0
    z = z * z + z0
    z = z * z + z0
    stdio.writeln(z)

if __name__ == '__main__':
    _main()

$ python complexpolar.py
-7.0 + 7.0i
def __abs__(self):
    return self._r

def __str__(self):
    return str(self.re()) + ' + ' + str(self.im()) + 'i'

def _main():
    z0 = Complex(1.0, 1.0)
    z = z0
    z = z * z + z0
    z = z * z + z0
   stdio.writeln(z)

if __name__ == '__main__':
    _main()

$ python complexpolar.py
-7.0 + 7.0i
### A data type `Counter` for counting

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Counter(id, maxCount)</code></td>
<td>a new counter <code>c</code> named <code>id</code>, with maximum value <code>maxCount</code></td>
</tr>
<tr>
<td><code>c.increment()</code></td>
<td>increment <code>c</code>, unless its value is <code>maxCount</code></td>
</tr>
<tr>
<td><code>c.value()</code></td>
<td>value of <code>c</code></td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of <code>c</code></td>
</tr>
<tr>
<td><code>c == d</code></td>
<td>are <code>c</code> and <code>d</code> equal?</td>
</tr>
<tr>
<td><code>c != d</code></td>
<td>are <code>c</code> and <code>d</code> not equal?</td>
</tr>
<tr>
<td><code>c &lt; d</code></td>
<td>is <code>c</code> less than <code>d</code>?</td>
</tr>
<tr>
<td><code>c &gt; d</code></td>
<td>is <code>c</code> greater than <code>d</code>?</td>
</tr>
<tr>
<td><code>c &lt;= d</code></td>
<td>is <code>c</code> less than or equal to <code>d</code>?</td>
</tr>
<tr>
<td><code>c &gt;= d</code></td>
<td>is <code>c</code> greater than or equal to <code>d</code>?</td>
</tr>
</tbody>
</table>
A data type `Counter` for counting

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Counter(id, maxCount)</code></td>
<td>a new counter <code>c</code> named <code>id</code>, with maximum value <code>maxCount</code></td>
</tr>
<tr>
<td><code>c.increment()</code></td>
<td>increment <code>c</code>, unless its value is <code>maxCount</code></td>
</tr>
<tr>
<td><code>c.value()</code></td>
<td>value of <code>c</code></td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of <code>c</code></td>
</tr>
<tr>
<td><code>c == d</code></td>
<td>are <code>c</code> and <code>d</code> equal?</td>
</tr>
<tr>
<td><code>c != d</code></td>
<td>are <code>c</code> and <code>d</code> not equal?</td>
</tr>
<tr>
<td><code>c &lt; d</code></td>
<td>is <code>c</code> less than <code>d</code>?</td>
</tr>
<tr>
<td><code>c &gt; d</code></td>
<td>is <code>c</code> greater than <code>d</code>?</td>
</tr>
<tr>
<td><code>c &lt;= d</code></td>
<td>is <code>c</code> less than or equal to <code>d</code>?</td>
</tr>
<tr>
<td><code>c &gt;= d</code></td>
<td>is <code>c</code> greater than or equal to <code>d</code>?</td>
</tr>
</tbody>
</table>
Counter.py: Definition of Counter data type.

```python
import stdio
import stdrandom
import sys

class Counter:
    def __init__(self, id, maxCount):
        self._name = id
        self._maxCount = maxCount
        self._count = 0

    def increment(self):
        if self._count < self._maxCount:
            self._count += 1

    def value(self):
        return self._count

    def __str__(self):
        return self._name + ': ' + str(self._count)

    def __eq__(self, other):
        return self._count == other._count

    def __ne__(self, other):
        return self._count != other._count

    def __lt__(self, other):
        return self._count < other._count
```
import stdio
import stdrandom
import sys

class Counter:
    def __init__(self, id, maxCount):
        self._name = id
        self._maxCount = maxCount
        self._count = 0

    def increment(self):
        if self._count < self._maxCount:
            self._count += 1

    def value(self):
        return self._count

    def __str__(self):
        return self._name + ': ' + str(self._count)

    def __eq__(self, other):
        return self._count == other._count

    def __ne__(self, other):
        return self._count != other._count

    def __lt__(self, other):
        return self._count < other._count
Examples

def __gt__(self, other):
    return self._count > other._count

def __le__(self, other):
    return self._count <= other._count

def __ge__(self, other):
    return self._count >= other._count

def _main():
    n = int(sys.argv[1])
    p = float(sys.argv[2])
    heads = Counter('Heads', n)
    tails = Counter('Tails', n)
    for i in range(n):
        if stdrandom.bernoulli(p):
            heads.increment()
        else:
            tails.increment()
    stdio.writeln(heads)
    stdio.writeln(tails)

if __name__ == '__main__':
    _main()

$ python counter.py 1000 .5
Heads: 483
Tails: 517
$ python counter.py 1000 .5
Heads: 503
Tails: 497
$ python counter.py 1000 .3
Heads: 280
Tails: 720
Examples

def __gt__(self, other):
    return self._count > other._count

def __le__(self, other):
    return self._count <= other._count

def __ge__(self, other):
    return self._count >= other._count

def _main():
    n = int(sys.argv[1])
    p = float(sys.argv[2])
    heads = Counter('Heads', n)
    tails = Counter('Tails', n)
    for i in range(n):
        if stdrandom.bernoulli(p):
            heads.increment()
        else:
            tails.increment()
    stdio.writeln(heads)
    stdio.writeln(tails)

if __name__ == '__main__':
    _main()

$ python counter.py 1000 .5
Heads: 483
Tails: 517
$ python counter.py 1000 .5
Heads: 503
Tails: 497
$ python counter.py 1000 .3
Heads: 280
Tails: 720
Examples

def __gt__(self, other):
    return self._count > other._count

def __le__(self, other):
    return self._count <= other._count

def __ge__(self, other):
    return self._count >= other._count

def _main():
    n = int(sys.argv[1])
    p = float(sys.argv[2])
    heads = Counter('Heads', n)
    tails = Counter('Tails', n)
    for i in range(n):
        if stdrandom.bernoulli(p):
            heads.increment()
        else:
            tails.increment()
    stdio.writeln(heads)
    stdio.writeln(tails)

if __name__ == '__main__':
    _main()

$ python counter.py 1000 .5
Heads: 483
Tails: 517
$ python counter.py 1000 .5
Heads: 503
Tails: 497
$ python counter.py 1000 .3
Heads: 280
Tails: 720
A spatial vector is an abstract entity that has a magnitude and a direction.

Vector operations, assuming \( \mathbf{x} = (x_1, x_2, \ldots, x_n) \), \( \mathbf{y} = (y_1, y_2, \ldots, y_n) \), and \( \alpha \in \mathbb{R} \):

- **Addition:** \( \mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \)
- **Subtraction:** \( \mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n) \)
- **Scalar product:** \( \alpha \mathbf{x} = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n) \)
- **Dot product:** \( \mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \)
- **Magnitude:** \( |\mathbf{x}| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \)
- **Direction:** \( \mathbf{x}/|\mathbf{x}| = (x_1/|\mathbf{x}|, x_2/|\mathbf{x}|, \ldots, x_n/|\mathbf{x}|) \)
A spatial vector is an abstract entity that has a magnitude and a direction.

Vector operations, assuming \( \mathbf{x} = (x_1, x_2, \ldots, x_n) \), \( \mathbf{y} = (y_1, y_2, \ldots, y_n) \), and \( \alpha \in \mathbb{R} \):

- **Addition:** \( \mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \)
- **Subtraction:** \( \mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n) \)
- **Scalar product:** \( \alpha \mathbf{x} = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n) \)
- **Dot product:** \( \mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \)
- **Magnitude:** \( |\mathbf{x}| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \)
- **Direction:** \( \mathbf{x}/|\mathbf{x}| = (x_1/|\mathbf{x}|, x_2/|\mathbf{x}|, \ldots, x_n/|\mathbf{x}|) \)
A spatial vector is an abstract entity that has a magnitude and a direction.

Vector operations, assuming \( \mathbf{x} = (x_1, x_2, \ldots, x_n) \), \( \mathbf{y} = (y_1, y_2, \ldots, y_n) \), and \( \alpha \in \mathbb{R} \)

- **Addition**: \( \mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \)
- **Subtraction**: \( \mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n) \)
- **Scalar product**: \( \alpha \mathbf{x} = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n) \)
- **Dot product**: \( \mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \)
- **Magnitude**: \( |\mathbf{x}| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \)
- **Direction**: \( \mathbf{x}/|\mathbf{x}| = (x_1/|\mathbf{x}|, x_2/|\mathbf{x}|, \ldots, x_n/|\mathbf{x}|) \)
A data type `Vector` for spatial vectors

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Vector(a)</code></td>
<td>a new vector $v$ with Cartesian coordinates taken from the list $a$</td>
</tr>
<tr>
<td>$v[i]$</td>
<td>$i$th Cartesian coordinates of $v$</td>
</tr>
<tr>
<td>$v + w$</td>
<td>sum of $v$ and $w$</td>
</tr>
<tr>
<td>$v - w$</td>
<td>difference of $v$ and $w$</td>
</tr>
<tr>
<td><code>v.scale(alpha)</code></td>
<td>scalar product of float $\alpha$ and $v$</td>
</tr>
<tr>
<td><code>v.dot(w)</code></td>
<td>dot product of $v$ and $w$</td>
</tr>
<tr>
<td><code>v.direction()</code></td>
<td>unit vector in the same direction as $v$</td>
</tr>
<tr>
<td><code>abs(v)</code></td>
<td>magnitude of $v$</td>
</tr>
<tr>
<td><code>len(v)</code></td>
<td>length of $v$</td>
</tr>
<tr>
<td><code>str(v)</code></td>
<td>string representation of $v$</td>
</tr>
</tbody>
</table>
A data type `Vector` for spatial vectors

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector(a)</td>
<td>a new vector $v$ with Cartesian coordinates taken from the list $a$</td>
</tr>
<tr>
<td>$v[i]$</td>
<td>$i$th Cartesian coordinates of $v$</td>
</tr>
<tr>
<td>$v + w$</td>
<td>sum of $v$ and $w$</td>
</tr>
<tr>
<td>$v - w$</td>
<td>difference of $v$ and $w$</td>
</tr>
<tr>
<td>$v$.scale(alpha)</td>
<td>scalar product of float $\alpha$ and $v$</td>
</tr>
<tr>
<td>$v$.dot(w)</td>
<td>dot product of $v$ and $w$</td>
</tr>
<tr>
<td>$v$.direction()</td>
<td>unit vector in the same direction as $v$</td>
</tr>
<tr>
<td>abs(v)</td>
<td>magnitude of $v$</td>
</tr>
<tr>
<td>len(v)</td>
<td>length of $v$</td>
</tr>
<tr>
<td>str(v)</td>
<td>string representation of $v$</td>
</tr>
</tbody>
</table>
import math
import stdarray
import stdio

class Vector:
    def __init__(self, a):
        self._coords = a[:]
        self._n = len(a)

    def __getitem__(self, i):
        return self._coords[i]

    def __add__(self, other):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = self._coords[i] + other._coords[i]
        return Vector(result)

    def __sub__(self, other):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = self._coords[i] - other._coords[i]
        return Vector(result)

    def scale(self, alpha):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = alpha * self._coords[i]
        return Vector(result)
Examples

vector.py: Definition of Vector data type.

```python
import math
import stdarray
import stdio

class Vector:
    def __init__(self, a):
        self._coords = a[:]
        self._n = len(a)

    def __getitem__(self, i):
        return self._coords[i]

    def __add__(self, other):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = self._coords[i] + other._coords[i]
        return Vector(result)

    def __sub__(self, other):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = self._coords[i] - other._coords[i]
        return Vector(result)

    def scale(self, alpha):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = alpha * self._coords[i]
        return Vector(result)
```
Examples

def dot(self, other):
    result = 0
    for i in range(self._n):
        result += self._coords[i] * other._coords[i]
    return result

def direction(self):
    return self.scale(1.0 / abs(self))

def __abs__(self):
    return math.sqrt(self.dot(self))

def __len__(self):
    return self._n

def __str__(self):
    return str(self._coords)

def _main():
    xCoords = [1.0, 2.0, 3.0, 4.0]
    yCoords = [5.0, 2.0, 4.0, 1.0]
    x = Vector(xCoords)
    y = Vector(yCoords)
    stdio.writeln('x = ' + str(x))
    stdio.writeln('y = ' + str(y))
    stdio.writeln('x + y = ' + str(x + y))
    stdio.writeln('10x = ' + str(x.scale(10.0)))
    stdio.writeln('|x| = ' + str(abs(x)))
    stdio.writeln('<x, y> = ' + str(x.dot(y)))
    stdio.writeln('|x - y| = ' + str(abs(x - y)))

if __name__ == '__main__':
    _main()
Examples

def dot(self, other):
    result = 0
    for i in range(self._n):
        result += self._coords[i] * other._coords[i]
    return result

def direction(self):
    return self.scale(1.0 / abs(self))

def __abs__(self):
    return math.sqrt(self.dot(self))

def __len__(self):
    return self._n

def __str__(self):
    return str(self._coords)

def _main():
    xCoords = [1.0, 2.0, 3.0, 4.0]
    yCoords = [5.0, 2.0, 4.0, 1.0]
    x = Vector(xCoords)
    y = Vector(yCoords)
   stdio.writeln('x = ' + str(x))
   stdio.writeln('y = ' + str(y))
   stdio.writeln('x + y = ' + str(x + y))
   stdio.writeln('10x = ' + str(x.scale(10.0)))
   stdio.writeln('|x| = ' + str(abs(x)))
   stdio.writeln('<x, y> = ' + str(x.dot(y)))
   stdio.writeln('|x - y| = ' + str(abs(x - y)))

if __name__ == '__main__':
    _main()
Examples

$ python vector.py
x = [1.0, 2.0, 3.0, 4.0]
y = [5.0, 2.0, 4.0, 1.0]
x + y = [6.0, 4.0, 7.0, 5.0]
10x = [10.0, 20.0, 30.0, 40.0]
|x| = 5.47722557505
<x, y> = 25.0
|x - y| = 5.09901951359
Examples

$ python vector.py
x = [1.0, 2.0, 3.0, 4.0]
y = [5.0, 2.0, 4.0, 1.0]
x + y = [6.0, 4.0, 7.0, 5.0]
10x = [10.0, 20.0, 30.0, 40.0]
|x| = 5.47722557505
<x, y> = 25.0
|x - y| = 5.09901951359
A data type `Sketch` for compactly representing the content of a document

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sketch(text, k, d)</td>
<td>a new sketch ( s ) built from the string ( text ) using ( k )-grams and dimension ( d )</td>
</tr>
<tr>
<td>s.similarTo(t)</td>
<td>similarity measure between sketches ( s ) and ( t ) (a float between 0.0 and 1.0)</td>
</tr>
<tr>
<td>str(s)</td>
<td>string representation of ( s )</td>
</tr>
</tbody>
</table>
A data type $\text{Sketch}$ for compactly representing the content of a document

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Sketch}(\text{text}, k, d)$</td>
<td>a new sketch $s$ built from the string $\text{text}$ using $k$-grams and dimension $d$</td>
</tr>
<tr>
<td>$s.\text{similarTo}(t)$</td>
<td>similarity measure between sketches $s$ and $t$ (a float between 0.0 and 1.0)</td>
</tr>
<tr>
<td>$\text{str}(s)$</td>
<td>string representation of $s$</td>
</tr>
</tbody>
</table>
import stdarray
import stdio
import sys
from vector import Vector

class Sketch:
    def __init__(self, text, k, d):
        freq = stdarray.create1D(d, 0)
        for i in range(len(text) - k):
            kgram = text[i:i + k]
            h = hash(kgram)
            freq[h % d] += 1
        vector = Vector(freq)
        self._sketch = vector.direction()

    def similarTo(self, other):
        return self._sketch.dot(other._sketch)

    def __str__(self):
        return str(self._sketch)

def _main():
    text = stdio.readAll()
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    sketch = Sketch(text, k, d)
    stdio.writeln(sketch)

if __name__ == '__main__':
    _main()
import stdarray
import stdio
import sys
from vector import Vector

class Sketch:
    def __init__(self, text, k, d):
        freq = stdarray.create1D(d, 0)
        for i in range(len(text) - k):
            kgram = text[i:i + k]
            h = hash(kgram)
            freq[h % d] += 1
        vector = Vector(freq)
        self._sketch = vector.direction()

    def similarTo(self, other):
        return self._sketch.dot(other._sketch)

    def __str__(self):
        return str(self._sketch)

def _main():
    text = stdio.readAll()
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    sketch = Sketch(text, k, d)
    stdio.writeln(sketch)

if __name__ == '__main__':
    _main()
Examples

$ more genome20.txt
ATAGATGCATAGCGCATAGC

$ python sketch.py 2 16 < genome20.txt
[0.37210420376762543, 0.37210420376762543, 0.49613893835683387, 0.0, 0.12403473458920847, 0.0, 0.0, 0.0, 0.0, 0.0, 0.24806946917841693, 0.0, 0.12403473458920847, 0.6201736729460423, 0.0, 0.0]
Examples

$ more genome20.txt
ATAGATGCATAGCGCATAGC

$ python sketch.py 2 16 < genome20.txt
[0.37210420376762543, 0.37210420376762543, 0.49613893835683387, 0.0,
 0.12403473458920847, 0.0, 0.0, 0.0, 0.0, 0.0, 0.24806946917841693, 0.0,
 0.12403473458920847, 0.62017367294617841693, 0.0, 0.0]
Examples

$ more genome20.txt
ATAGATGCATAGCGCATAGC

$ python sketch.py 2 16 < genome20.txt
[0.37210420376762543, 0.37210420376762543, 0.49613893835683387, 0.0, 0.12403473458920847, 0.0, 0.0, 0.0, 0.0, 0.0, 0.24806946917841693, 0.0, 0.12403473458920847, 0.620173672946917841693, 0.0, 0.0]
Examples

comparedocuments.py: Accept integers \( k \) and \( d \) as command-line arguments, read a document list from standard input, compute \( d \)-dimensional profiles based on \( k \)-gram frequencies for all the documents, and write a matrix of similarity measures between all pairs of documents.

```python
import stdarray
import stdio
import sys
from instream import InStream
from sketch import Sketch

def main():
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    filenames = stdio.readAllStrings()
    sketches = stdarray.create1D(len(filenames))
    for i in range(len(filenames)):
        text = InStream(filenames[i]).readAll()
        sketches[i] = Sketch(text, k, d)
    stdio.write(' ')
    for i in range(len(filenames)):
        stdio.writef(' %8.4s ', filenames[i])
    stdio.writeln()
    for i in range(len(filenames)):
        stdio.writef('%8.4s', filenames[i])
    stdio.writeln()
    for i in range(len(filenames)):
        for j in range(len(filenames)):
            stdio.writef('%8.2f', sketches[i].similarTo(sketches[j]))
        stdio.writeln()

if __name__ == '__main__':
    main()
```
comparedocuments.py: Accept integers $k$ and $d$ as command-line arguments, read a document list from standard input, compute $d$-dimensional profiles based on $k$-gram frequencies for all the documents, and write a matrix of similarity measures between all pairs of documents.

```python
import stdarray
import stdio
import sys
from instream import InStream
from sketch import Sketch

def main():
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    filenames = stdio.readAllStrings()
    sketches = stdarray.create1D(len(filenames))
    for i in range(len(filenames)):
        text = InStream(filenames[i]).readAll()
        sketches[i] = Sketch(text, k, d)
    stdio.writeln()
    for i in range(len(filenames)):
        stdio.writeln('%8.4s' % filenames[i])
    for i in range(len(filenames)):
        for j in range(len(filenames)):
            stdio.writeln('%8.2f' % sketches[i].similarTo(sketches[j]))

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

```bash
$ more documents.txt
constitution.txt
tomsawyer.txt
huckfinn.txt
prejudice.txt
vector.py
djia.csv
amazon.html
actg.txt

$ python comparedocuments.py 5 10000 < documents.txt
```

<table>
<thead>
<tr>
<th></th>
<th>cons</th>
<th>toms</th>
<th>huck</th>
<th>prej</th>
<th>vect</th>
<th>djia</th>
<th>amaz</th>
<th>actg</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>1.00</td>
<td>0.67</td>
<td>0.61</td>
<td>0.64</td>
<td>0.10</td>
<td>0.18</td>
<td>0.19</td>
<td>0.12</td>
</tr>
<tr>
<td>toms</td>
<td>0.67</td>
<td>1.00</td>
<td>0.93</td>
<td>0.87</td>
<td>0.08</td>
<td>0.23</td>
<td>0.19</td>
<td>0.15</td>
</tr>
<tr>
<td>huck</td>
<td>0.61</td>
<td>0.93</td>
<td>1.00</td>
<td>0.81</td>
<td>0.06</td>
<td>0.21</td>
<td>0.15</td>
<td>0.14</td>
</tr>
<tr>
<td>prej</td>
<td>0.64</td>
<td>0.87</td>
<td>0.81</td>
<td>1.00</td>
<td>0.07</td>
<td>0.25</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td>vect</td>
<td>0.10</td>
<td>0.08</td>
<td>0.06</td>
<td>0.07</td>
<td>1.00</td>
<td>0.03</td>
<td>0.17</td>
<td>0.01</td>
</tr>
<tr>
<td>djia</td>
<td>0.18</td>
<td>0.23</td>
<td>0.21</td>
<td>0.25</td>
<td>0.03</td>
<td>1.00</td>
<td>0.13</td>
<td>0.12</td>
</tr>
<tr>
<td>amaz</td>
<td>0.19</td>
<td>0.19</td>
<td>0.15</td>
<td>0.19</td>
<td>0.17</td>
<td>0.13</td>
<td>1.00</td>
<td>0.09</td>
</tr>
<tr>
<td>actg</td>
<td>0.12</td>
<td>0.15</td>
<td>0.14</td>
<td>0.16</td>
<td>0.01</td>
<td>0.12</td>
<td>0.09</td>
<td>1.00</td>
</tr>
</tbody>
</table>
$ more documents.txt
constitution.txt
tomsawyer.txt
huckfinn.txt
prejudice.txt
vector.py
djia.csv
amazon.html
actg.txt

$ python comparedocuments.py 5 10000 < documents.txt

<table>
<thead>
<tr>
<th></th>
<th>cons</th>
<th>toms</th>
<th>huck</th>
<th>prej</th>
<th>vect</th>
<th>djia</th>
<th>amaz</th>
<th>actg</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>1.00</td>
<td>0.67</td>
<td>0.61</td>
<td>0.64</td>
<td>0.10</td>
<td>0.18</td>
<td>0.19</td>
<td>0.12</td>
</tr>
<tr>
<td>toms</td>
<td>0.67</td>
<td>1.00</td>
<td>0.93</td>
<td>0.87</td>
<td>0.08</td>
<td>0.23</td>
<td>0.19</td>
<td>0.15</td>
</tr>
<tr>
<td>huck</td>
<td>0.61</td>
<td>0.93</td>
<td>1.00</td>
<td>0.81</td>
<td>0.06</td>
<td>0.21</td>
<td>0.15</td>
<td>0.14</td>
</tr>
<tr>
<td>prej</td>
<td>0.64</td>
<td>0.87</td>
<td>0.81</td>
<td>1.00</td>
<td>0.07</td>
<td>0.25</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td>vect</td>
<td>0.10</td>
<td>0.08</td>
<td>0.06</td>
<td>0.07</td>
<td>1.00</td>
<td>0.03</td>
<td>0.17</td>
<td>0.01</td>
</tr>
<tr>
<td>djia</td>
<td>0.18</td>
<td>0.23</td>
<td>0.21</td>
<td>0.25</td>
<td>0.03</td>
<td>1.00</td>
<td>0.13</td>
<td>0.12</td>
</tr>
<tr>
<td>amaz</td>
<td>0.19</td>
<td>0.19</td>
<td>0.15</td>
<td>0.19</td>
<td>0.17</td>
<td>0.13</td>
<td>1.00</td>
<td>0.09</td>
</tr>
<tr>
<td>actg</td>
<td>0.12</td>
<td>0.15</td>
<td>0.14</td>
<td>0.16</td>
<td>0.01</td>
<td>0.12</td>
<td>0.09</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Examples

```bash
$ more documents.txt
constitution.txt
tomsawyer.txt
huckfinn.txt
prejudice.txt
vector.py
djia.csv
amazon.html
actg.txt

$ python comparedocuments.py 5 10000 < documents.txt

<table>
<thead>
<tr>
<th></th>
<th>cons</th>
<th>toms</th>
<th>huck</th>
<th>prej</th>
<th>vect</th>
<th>djia</th>
<th>amaz</th>
<th>actg</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>1.00</td>
<td>0.67</td>
<td>0.61</td>
<td>0.64</td>
<td>0.10</td>
<td>0.18</td>
<td>0.19</td>
<td>0.12</td>
</tr>
<tr>
<td>toms</td>
<td>0.67</td>
<td>1.00</td>
<td>0.93</td>
<td>0.87</td>
<td>0.08</td>
<td>0.23</td>
<td>0.19</td>
<td>0.15</td>
</tr>
<tr>
<td>huck</td>
<td>0.61</td>
<td>0.93</td>
<td>1.00</td>
<td>0.81</td>
<td>0.06</td>
<td>0.21</td>
<td>0.15</td>
<td>0.14</td>
</tr>
<tr>
<td>prej</td>
<td>0.64</td>
<td>0.87</td>
<td>0.81</td>
<td>1.00</td>
<td>0.07</td>
<td>0.25</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td>vect</td>
<td>0.10</td>
<td>0.08</td>
<td>0.06</td>
<td>0.07</td>
<td>1.00</td>
<td>0.03</td>
<td>0.17</td>
<td>0.01</td>
</tr>
<tr>
<td>djia</td>
<td>0.18</td>
<td>0.23</td>
<td>0.21</td>
<td>0.25</td>
<td>0.03</td>
<td>1.00</td>
<td>0.13</td>
<td>0.12</td>
</tr>
<tr>
<td>amaz</td>
<td>0.19</td>
<td>0.19</td>
<td>0.15</td>
<td>0.19</td>
<td>0.17</td>
<td>0.13</td>
<td>1.00</td>
<td>0.09</td>
</tr>
<tr>
<td>actg</td>
<td>0.12</td>
<td>0.15</td>
<td>0.14</td>
<td>0.16</td>
<td>0.01</td>
<td>0.12</td>
<td>0.09</td>
<td>1.00</td>
</tr>
</tbody>
</table>
```
Examples

A comparable data type `Planet` for representing planets

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Planet(name, moons)</code></td>
<td>create a new planet <code>p</code> given its name and number of moons</td>
</tr>
<tr>
<td><code>cmp(p, q)</code></td>
<td>compare planets <code>p</code> and <code>q</code> by name (return negative integer, zero, or positive integer depending on whether the name of <code>p</code> is less than, equal to, or greater than the name of <code>q</code>)</td>
</tr>
<tr>
<td><code>str(p)</code></td>
<td>string representation of <code>p</code></td>
</tr>
</tbody>
</table>
A comparable data type `Planet` for representing planets

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Planet(name, moons)</code></td>
<td>create a new planet $p$ given its name and number of moons</td>
</tr>
<tr>
<td><code>cmp(p, q)</code></td>
<td>compare planets $p$ and $q$ by name (return negative integer, zero, or positive integer depending on whether the name of $p$ is less than, equal to, or greater than the name of $q$)</td>
</tr>
<tr>
<td><code>str(p)</code></td>
<td>string representation of $p$</td>
</tr>
</tbody>
</table>
```python
import stdio

class Planet:
    def __init__(self, name, moons):
        self._name = name
        self._moons = moons

    def __cmp__(self, other):
        return cmp(self._name, other._name)

    def __str__(self):
        return self._name + ', ' + str(self._moons)
```

Examples

planet.py: Definition of Planet data type.
import stdio

class Planet:
    def __init__(self, name, moons):
        self._name = name
        self._moons = moons

    def __cmp__(self, other):
        return cmp(self._name, other._name)

    def __str__(self):
        return self._name + ', ' + str(self._moons)
def _main():
    planets = [None] * 8
    planets[0] = Planet("Mercury", 0)
    planets[1] = Planet("Venus", 0)
    planets[2] = Planet("Earth", 1)
    planets[3] = Planet("Mars", 2)
    planets[6] = Planet("Uranus", 27)
    planets[7] = Planet("Neptune", 14)
    stdio.writeln("Unsorted:")
    for v in planets:
        stdio.writeln("  " + str(v))
    planets.sort()
    stdio.writeln("Sorted by name:")
    for v in planets:
        stdio.writeln("  " + str(v))
    planets.sort(cmp = lambda x, y: cmp(x._moons, y._moons))
    stdio.writeln("Sorted by # of moons:")
    for v in planets:
        stdio.writeln("  " + str(v))

if __name__ == '__main__':
    _main()
def _main():
    planets = [None] * 8
    planets[0] = Planet("Mercury", 0)
    planets[1] = Planet("Venus", 0)
    planets[2] = Planet("Earth", 1)
    planets[3] = Planet("Mars", 2)
    planets[6] = Planet("Uranus", 27)
    planets[7] = Planet("Neptune", 14)
    stdin.writeln("Unsorted:")
    for v in planets:
        stdin.writeln(" " + str(v))
    planets.sort()
    stdin.writeln("Sorted by name:")
    for v in planets:
        stdin.writeln(" " + str(v))
    planets.sort(cmp = lambda x, y: cmp(x._moons, y._moons))
    stdin.writeln("Sorted by # of moons:")
    for v in planets:
        stdin.writeln(" " + str(v))

if __name__ == '__main__':
    _main()
$ python planet.py

Unsorted:
  Mercury, 0
  Venus, 0
  Earth, 1
  Mars, 2
  Jupiter, 67
  Saturn, 62
  Uranus, 27
  Neptune, 14

Sorted by name:
  Earth, 1
  Jupiter, 67
  Mars, 2
  Mercury, 0
  Neptune, 14
  Saturn, 62
  Uranus, 27
  Venus, 0

Sorted by # of moons:
  Mercury, 0
  Venus, 0
  Earth, 1
  Mars, 2
  Neptune, 14
  Uranus, 27
  Saturn, 62
  Jupiter, 67
Examples

$ python planet.py

Unsorted:
    Mercury, 0
    Venus, 0
    Earth, 1
    Mars, 2
    Jupiter, 67
    Saturn, 62
    Uranus, 27
    Neptune, 14

Sorted by name:
    Earth, 1
    Jupiter, 67
    Mars, 2
    Mercury, 0
    Neptune, 14
    Saturn, 62
    Uranus, 27
    Venus, 0

Sorted by # of moons:
    Mercury, 0
    Venus, 0
    Earth, 1
    Mars, 2
    Neptune, 14
    Uranus, 27
    Saturn, 62
    Jupiter, 67
Examples

An iterable Fibonacci data type for iterating over Fibonacci sequences

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(n)</td>
<td>a new object $f$ for iterating over the first $n$ Fibonacci numbers</td>
</tr>
<tr>
<td>iter(f)</td>
<td>an iterable object $fiter$ on $f$</td>
</tr>
<tr>
<td>next(fiter)</td>
<td>the next number in the Fibonacci sequence $fiter$</td>
</tr>
</tbody>
</table>
An iterable Fibonacci data type for iterating over Fibonacci sequences

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(n)</td>
<td>a new object ( f ) for iterating over the first ( n ) Fibonacci numbers</td>
</tr>
<tr>
<td>iter(f)</td>
<td>an iterable object ( fiter ) on ( f )</td>
</tr>
<tr>
<td>next(fiter)</td>
<td>the next number in the Fibonacci sequence ( fiter )</td>
</tr>
</tbody>
</table>
Examples

fibonacci.py: Definition of Fibonacci data type.

```python
import stdio
import sys

class Fibonacci:
    def __init__(self, n):
        self._n = n
        self._current = 0
        self._a = 1
        self._b = 1

    def __iter__(self):
        return self

    def next(self):
        self._current += 1
        if self._current > self._n:
            raise StopIteration
        if self._current <= 2:
            return 1
        self._a, self._b = self._b, self._a + self._b
        return self._b

def _main():
    n = int(sys.argv[1])
    for i in Fibonacci(n):
        stdio.writeln(i)

if __name__ == '__main__':
    _main()
```
import stdio
import sys

class Fibonacci:
    def __init__(self, n):
        self._n = n
        self._current = 0
        self._a = 1
        self._b = 1

    def __iter__(self):
        return self

    def next(self):
        self._current += 1
        if self._current > self._n:
            raise StopIteration
        if self._current <= 2:
            return 1
        self._a, self._b = self._b, self._a + self._b
        return self._b

def _main():
    n = int(sys.argv[1])
    for i in Fibonacci(n):
        stdio.writeln(i)

if __name__ == '__main__':
    _main()
Examples

```python
from fibonacci import Fibonacci
>>> f = Fibonacci(5)
>>> fiter = iter(f)  # calls f.__iter__()
>>> next(fiter)      # calls fiter.next()
1
>>> next(fiter)
1
>>> next(fiter)
2
>>> next(fiter)
3
>>> next(fiter)
5
>>> next(fiter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

$ python fibonacci.py 10
1
1
2
3
5
8
13
21
34
55
```
Examples

```python
>>> from fibonacci import Fibonacci
>>> f = Fibonacci(5)
>>> fiter = iter(f)  # calls f.__iter__()
>>> next(fiter)  # calls fiter.next()
1
>>> next(fiter)
1
>>> next(fiter)
2
>>> next(fiter)
3
>>> next(fiter)
5
>>> next(fiter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```sh
$ python fibonacci.py 10
1
1
2
3
5
8
13
21
34
55
```
Examples

```python
>>> from fibonacci import Fibonacci
>>> f = Fibonacci(5)
>>> fiter = iter(f)  # calls f.__iter__()
>>> next(fiter)  # calls fitter.next()
1
>>> next(fiter)
1
>>> next(fiter)
2
>>> next(fiter)
3
>>> next(fiter)
5
>>> next(fiter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
$ python fibonacci.py 10
1
1
2
3
5
8
13
21
34
55
```
Inheritance

Python provides language support for defining relationships among classes, known as inheritance.

Inheritance enables subclassing, where the idea is to define a new class (subclass, or derived class) that inherits instance variables and methods from another class (superclass, or base class).

class DerivedClassName(BaseClassName):
   <statement>
   <statement>
   ...

Every class in Python implicitly inherits from object.

Python supports two built-in functions that work with inheritance:

- `isinstance(o, T)` checks if instance `o` is of type `T`.
- `issubclass(T1, T2)` checks if `T1` is a subclass of `T2`.

Python supports a limited form of multiple inheritance as well.

class DerivedClassName(Base1, Base2, Base3):
   <statement>
   <statement>
   ...

Inheritance

Python provides language support for defining relationships among classes, known as inheritance

Inheritance enables subclassing, where the idea is to define a new class (subclass, or derived class) that inherits instance variables and methods from another class (superclass, or base class)

```python
class DerivedClassName(BaseClassName):
    <statement>
    <statement>
    ...
```

Every class in Python implicitly inherits from `object`

Python supports two built-in functions that work with inheritance

- `isinstance(o, T)` checks if instance `o` is of type `T`
- `issubclass(T1, T2)` checks if `T1` is a subclass of `T2`

Python supports a limited form of multiple inheritance as well

```python
class DerivedClassName(Base1, Base2, Base3):
    <statement>
    <statement>
    ...
```
Inheritance

Python provides language support for defining relationships among classes, known as inheritance

Inheritance enables subclassing, where the idea is to define a new class (subclass, or derived class) that inherits instance variables and methods from another class (superclass, or base class)

class DerivedClassName(BaseClassName):
    <statement>
    <statement>
    ...

Every class in Python implicitly inherits from object

Python supports two built-in functions that work with inheritance

- `isinstance(o, T)` checks if instance `o` is of type `T`
- `issubclass(T1, T2)` checks if `T1` is a subclass of `T2`

Python supports a limited form of multiple inheritance as well

class DerivedClassName(Base1, Base2, Base3):
    <statement>
    <statement>
    ...

Inheritance

Python provides language support for defining relationships among classes, known as inheritance.

Inheritance enables subclassing, where the idea is to define a new class (subclass, or derived class) that inherits instance variables and methods from another class (superclass, or base class).

class DerivedClassName(BaseClassName):
    <statement>
    <statement>
    ...

Every class in Python implicitly inherits from `object`.

Python supports two built-in functions that work with inheritance:

- `isinstance(o, T)` checks if instance `o` is of type `T`
- `issubclass(T1, T2)` checks if `T1` is a subclass of `T2`

Python supports a limited form of multiple inheritance as well.

class DerivedClassName(Base1, Base2, Base3):
    <statement>
    <statement>
    ...

Inheritance

Python provides language support for defining relationships among classes, known as inheritance.

Inheritance enables subclassing, where the idea is to define a new class (subclass, or derived class) that inherits instance variables and methods from another class (superclass, or base class).

class DerivedClassName(BaseClassName):
    <statement>
    <statement>
    ...

Every class in Python implicitly inherits from `object`.

Python supports two built-in functions that work with inheritance:

- `isinstance(o, T)` checks if instance `o` is of type `T`
- `issubclass(T1, T2)` checks if `T1` is a subclass of `T2`

Python supports a limited form of multiple inheritance as well.

class DerivedClassName(Base1, Base2, Base3):
    <statement>
    <statement>
    ...

34 / 37
Inheritance

Python provides language support for defining relationships among classes, known as inheritance

Inheritance enables subclassing, where the idea is to define a new class (subclass, or derived class) that inherits instance variables and methods from another class (superclass, or base class)

```python
class DerivedClassName(BaseClassName):
    <statement>
    <statement>
    ...
```

Every class in Python implicitly inherits from `object`

Python supports two built-in functions that work with inheritance

- `isinstance(o, T)` checks if instance `o` is of type `T`
- `issubclass(T1, T2)` checks if `T1` is a subclass of `T2`

Python supports a limited form of multiple inheritance as well

```python
class DerivedClassName(Base1, Base2, Base3):
    <statement>
    <statement>
    ...
```
Design by contract model is one in which the designer of a data type expresses

- A precondition - the condition that the client promises to satisfy when calling a method
- A postcondition - the condition that the implementation promises to achieve when returning from a method
- Invariants - any condition that the implementation promises to satisfy while the method is executing
- Side effects - any other change in state that the method could cause

Exceptions and assertions are Python language mechanisms that enable us to test these conditions
Design by Contract

Design by contract model is one in which the designer of a data type expresses

- A precondition - the condition that the client promises to satisfy when calling a method
- A postcondition - the condition that the implementation promises to achieve when returning from a method
- Invariants - any condition that the implementation promises to satisfy while the method is executing
- Side effects - any other change in state that the method could cause

Exceptions and assertions are Python language mechanisms that enable us to test these conditions
Design by contract model is one in which the designer of a data type expresses

• A precondition - the condition that the client promises to satisfy when calling a method

• A postcondition - the condition that the implementation promises to achieve when returning from a method

• Invariants - any condition that the implementation promises to satisfy while the method is executing

• Side effects - any other change in state that the method could cause

Exceptions and assertions are Python language mechanisms that enable us to test these conditions
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error.

The action taken in response is known as raising an exception (or error).

We can raise our own exceptions as follows:

```python
raise Exception('Error message here.')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions:

```python
if len(self) != len(other):
    raise Exception('Vectors have different dimensions')
```

We can handle exceptions using a try-except block.

For example:

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error.

The action taken in response is known as raising an exception (or error).

We can raise our own exceptions as follows:

```python
raise Exception('Error message here.')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions:

```python
if len(self) != len(other):
    raise Exception('Vectors have different dimensions')
```

We can handle exceptions using a try-except block:

For example:

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error

The action taken in response is known as raising an exception (or error)

We can raise our own exceptions as follows

```python
raise Exception('Error message here.')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions

```python
if len(self) != len(other):
    raise Exception('Vectors have different dimensions')
```

We can handle exceptions using a try-except block

For example

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error.

The action taken in response is known as raising an exception (or error).

We can raise our own exceptions as follows:

```python
raise Exception('Error message here.')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions:

```python
if len(self) != len(other):
    raise Exception('vectors have different dimensions')
```

We can handle exceptions using a try-except block.

For example:

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error.

The action taken in response is known as raising an exception (or error).

We can raise our own exceptions as follows:

```python
raise Exception('Error message here. ')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions:

```python
if len(self) != len(other):
    raise Exception('vectors have different dimensions')
```

We can handle exceptions using a `try-except` block.

For example:

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error

The action taken in response is known as raising an exception (or error)

We can raise our own exceptions as follows

```python
raise Exception('Error message here.')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions

```python
if len(self) != len(other):
    raise Exception('vectors have different dimensions')
```

We can handle exceptions using a try-except block

For example

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
Design by Contract

An exception is a disruptive event that occurs while a program is running, often to signal an error.

The action taken in response is known as raising an exception (or error).

We can raise our own exceptions as follows:

```python
raise Exception('Error message here.')
```

For example, in `vector.py`, we can raise an exception in `__add__()` if the two `Vectors` to be added have different dimensions:

```python
if len(self) != len(other):
    raise Exception('Vectors have different dimensions')
```

We can handle exceptions using a try-except block.

For example:

```python
(x, y) = (5, 0)
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
stdio.writeln(z)
```
An assertion is a boolean expression that we affirm is True, and if it is False, the program will raise an AssertionError at run time.

For example, in `counter.py`, we might check that the counter is never negative by adding the following assertion as the last statement in `increment()`:

```python
assert self.__count >= 0
```

We can also include an optional message, such as:

```python
assert self.__count >= 0, 'Negative count detected'
```
An assertion is a boolean expression that we affirm is True, and if it is False, the program will raise an AssertionError at run time.

For example, in `counter.py`, we might check that the counter is never negative by adding the following assertion as the last statement in `increment()`:

```python
assert self.__count >= 0
```

We can also include an optional message, such as:

```python
assert self.__count >= 0, 'Negative count detected'
```
An assertion is a boolean expression that we affirm is True, and if it is False, the program will raise an AssertionError at run time.

For example, in counter.py, we might check that the counter is never negative by adding the following assertion as the last statement in increment():

```python
assert self.__count >= 0
```

We can also include an optional message, such as:

```python
assert self.__count >= 0, 'Negative count detected'
```
An assertion is a boolean expression that we affirm is True, and if it is False, the program will raise an AssertionError at run time.

For example, in counter.py, we might check that the counter is never negative by adding the following assertion as the last statement in increment():

```python
assert self.__count >= 0
```

We can also include an optional message, such as:

```python
assert self.__count >= 0, 'Negative count detected'
```