Designing Data Types
Outline

1 APIs
2 Encapsulation
3 Immutability
4 Polymorphism
5 Overloading
6 Functions are Objects
7 Examples
8 Exceptions
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.
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.
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.
The process of separating clients from implementations by hiding information is known as encapsulation.
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

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

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.
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.
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.
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><code>__add__</code></td>
<td>sum of $x$ and $y$</td>
</tr>
<tr>
<td>$x - y$</td>
<td><code>__sub__</code></td>
<td>difference of $x$ and $y$</td>
</tr>
<tr>
<td>$x \times y$</td>
<td><code>__mul__</code></td>
<td>product of $x$ and $y$</td>
</tr>
<tr>
<td>$x^y$</td>
<td><code>__pow__</code></td>
<td>$x$ to the power $y$</td>
</tr>
<tr>
<td>$x / y$</td>
<td><code>__div__</code></td>
<td>quotient of $x$ and $y$</td>
</tr>
<tr>
<td>$x \div y$</td>
<td><code>__floordiv__</code></td>
<td>floored quotient of $x$ and $y$</td>
</tr>
<tr>
<td>$x % y$</td>
<td><code>__mod__</code></td>
<td>remainder when dividing $x$ by $y$</td>
</tr>
<tr>
<td>$+x$</td>
<td><code>__pos__</code></td>
<td>$x$</td>
</tr>
<tr>
<td>$-x$</td>
<td><code>__neg__</code></td>
<td>arithmetic negation of $x$</td>
</tr>
</tbody>
</table>
### 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><code>__add__(self, y)</code></td>
<td>sum of $x$ and $y$</td>
</tr>
<tr>
<td>$x - y$</td>
<td><code>__sub__(self, y)</code></td>
<td>difference of $x$ and $y$</td>
</tr>
<tr>
<td>$x \times y$</td>
<td><code>__mul__(self, y)</code></td>
<td>product of $x$ and $y$</td>
</tr>
<tr>
<td>$x^y$</td>
<td><code>__pow__(self, y)</code></td>
<td>$x$ to the power $y$</td>
</tr>
<tr>
<td>$x / y$</td>
<td><code>__div__(self, y)</code></td>
<td>quotient of $x$ and $y$</td>
</tr>
<tr>
<td>$x \div y$</td>
<td><code>__floordiv__(self, y)</code></td>
<td>floored quotient of $x$ and $y$</td>
</tr>
<tr>
<td>$x % y$</td>
<td><code>__mod__(self, y)</code></td>
<td>remainder when dividing $x$ by $y$</td>
</tr>
<tr>
<td>$+x$</td>
<td><code>__pos__(self)</code></td>
<td>$x$</td>
</tr>
<tr>
<td>$-x$</td>
<td><code>__neg__(self)</code></td>
<td>arithmetic negation of $x$</td>
</tr>
<tr>
<td>Client Operation</td>
<td>Special Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<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 comparison operators

<table>
<thead>
<tr>
<th>Client Operation</th>
<th>Special Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == y</code></td>
<td><code>__eq__(self, y)</code></td>
<td>are x and y equal?</td>
</tr>
<tr>
<td><code>x != y</code></td>
<td><code>__ne__(self, y)</code></td>
<td>are x and y not equal?</td>
</tr>
<tr>
<td><code>x &lt; y</code></td>
<td><code>__lt__(self, y)</code></td>
<td>is x less than y?</td>
</tr>
<tr>
<td><code>x &lt;= y</code></td>
<td><code>__le__(self, y)</code></td>
<td>is x less than or equal to y?</td>
</tr>
<tr>
<td><code>x &gt; y</code></td>
<td><code>__gt__(self, y)</code></td>
<td>is x greater than y?</td>
</tr>
<tr>
<td><code>x &gt;= y</code></td>
<td><code>__ge__(self, y)</code></td>
<td>is x greater than or equal to y?</td>
</tr>
<tr>
<td>Client Operation</td>
<td>Special Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>length of x</td>
<td><strong>len</strong>(self)</td>
<td><strong>len</strong>(self)</td>
</tr>
<tr>
<td>float equivalent of x</td>
<td><strong>float</strong>(self)</td>
<td><strong>float</strong>(self)</td>
</tr>
<tr>
<td>integer equivalent of x</td>
<td><strong>int</strong>(self)</td>
<td><strong>int</strong>(self)</td>
</tr>
<tr>
<td>string representation of x</td>
<td><strong>str</strong>(self)</td>
<td><strong>str</strong>(self)</td>
</tr>
<tr>
<td>absolute value of x</td>
<td><strong>abs</strong>(self)</td>
<td><strong>abs</strong>(self)</td>
</tr>
<tr>
<td>integer hash code for x</td>
<td><strong>hash</strong>(self)</td>
<td><strong>hash</strong>(self)</td>
</tr>
<tr>
<td>iterator for x</td>
<td><strong>iter</strong>(self)</td>
<td><strong>iter</strong>(self)</td>
</tr>
<tr>
<td>Client Operation</td>
<td>Special Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>------------------------------</td>
</tr>
<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>
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)
```
In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results.
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.

```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

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 (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)
```
Examples

A complex number $z$ in the cartesian form is expressed as $z = x + yi$, where $x$ (the real part) and $y$ (the imaginary part) are real numbers and $i = \sqrt{-1}$.

**Complex arithmetic**

- **Conjugate**: $(x + yi) \star = x - yi$
- **Addition**: $(x + yi) + (v + wi) = (x + v) + (y + w)i$
- **Multiplication**: $(x + yi) \times (v + wi) = (xv - yw) + (yv + xw)i$
- **Magnitude**: $|x + yi| = \sqrt{x^2 + y^2}$
Examples

A complex number $z$ in the cartesian form is expressed as $z = x + yi$, where $x$ (the real part) and $y$ (the imaginary part) are real numbers and $i = \sqrt{-1}$.
Examples

A complex number \( z \) in the cartesian form is expressed as \( z = x + yi \), where \( x \) (the real part) and \( y \) (the imaginary part) are real numbers and \( i = \sqrt{-1} \).

Complex arithmetic

- Conjugate: \( (x + yi)^* = x - yi \)
- Addition: \( (x + yi) + (v + wi) = (x + v) + (y + w)i \)
- Multiplication: \( (x + yi) \times (v + wi) = (xv - yw) + (yw + xw)i \)
- Magnitude: \( |x + yi| = \sqrt{x^2 + y^2} \)
Examples

A data type `Complex` for representing complex numbers:

```python
Complex(x, y)
```

A new complex object `c` with value `x + yi`:

```python
c.re()  # real part of c
c.im()  # imaginary part of c
c.conjugate()  # conjugate of c
c + d  # sum of c and d
c * d  # product of c and d
c == d  # are c and d equal?
abs(c)  # magnitude of c
str(c)  # string representation of c
```
A data type `Complex` for representing 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>c == d</code></td>
<td>are <code>c</code> and <code>d</code> equal?</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>
import math
import stdio

class Complex:
    def __init__(self, re=0.0, im=0.0):
        self._re = re
        self._im = im
    def re(self):
        return self._re
    def im(self):
        return self._im
    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):
        re = self._re * other._re - self._im * other._im
        im = self._re * other._im + self._im * other._re
        return Complex(re, im)
    def __abs__(self):
        return math.sqrt(self._re * self._re + self._im * self._im)
    def __eq__(self, other):
        return self._re == other._re and self._im == other._im
    def __str__(self):
        SUFFIX = 'i'
```python
import math
import stdio

class Complex:
    def __init__(self, re=0.0, im=0.0):
        self._re = re
        self._im = im

    def re(self):
        return self._re

    def im(self):
        return self._im

    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):
        re = self._re * other._re - self._im * other._im
        im = self._re * other._im + self._im * other._re
        return Complex(re, im)

    def __abs__(self):
        return math.sqrt(self._re * self._re + self._im * self._im)

    def __eq__(self, other):
        return self._re == other._re and self._im == other._im

    def __str__(self):
        SUFFIX = 'i'
```

Examples

```python
if self._im == 0:
    return str(self._re)
elif self._re == 0:
    return str(self._im) + SUFFIX
elif self._im < 0:
    return str(self._re) + ' - ' + str(-self._im) + SUFFIX
else:
    return str(self._re) + ' + ' + str(self._im) + SUFFIX

def _main():
a = Complex(5.0, -6.0)
b = Complex(3.0, 4.0)
stdio.writeln("a = " + str(a))
stdio.writeln("b = " + str(b))
stdio.writeln("conj(a) = " + str((a.conjugate())))
stdio.writeln("a + b = " + str(a + b))
stdio.writeln("a * b = " + str(a * b))
stdio.writeln("|b| = " + str(abs(b)))

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

Program: mandelbrot.py
• Command-line input: xc (float), yc (float), and size (float)
• Standard draw output: size-by-size region of the Mandelbrot set, centered at (xc, yc)

/terminal
$ python3 mandelbrot.py -0.5 0 2

/terminal
$ python3 mandelbrot.py 0.1015 -0.633 .01
Examples

**Program:** mandelbrot.py

**Command-line input:**
- `xc` (float)
- `yc` (float)
- `size` (float)

**Standard draw output:**
- `size`-by-`size` region of the Mandelbrot set, centered at (`xc`, `yc`)
Examples

Program: mandelbrot.py
  • Command-line input: $xc$ (float), $yc$ (float), and $size$ (float)
Examples

Program: mandelbrot.py

- Command-line input: \(xc\) (float), \(yc\) (float), and \(size\) (float)
- Standard draw output: \(size\)-by-\(size\) region of the Mandelbrot set, centered at \((xc, yc)\)
Examples

Program: `mandelbrot.py`

- Command-line input: `xc` (float), `yc` (float), and `size` (float)
- Standard draw output: `size`-by-`size` region of the Mandelbrot set, centered at `(xc, yc)`

```
$ mkdir ~/workspace/ipp/programs
$ python3 mandelbrot.py -0.5 0 2
```
Examples

Program: mandelbrot.py

- Command-line input: \( xc \) (float), \( yc \) (float), and \( size \) (float)
- Standard draw output: size-by-size region of the Mandelbrot set, centered at \( (xc, yc) \)

```bash
> ~/workspace/ipp/programs
$ python3 mandelbrot.py -0.5 0 2
```

```bash
> ~/workspace/ipp/programs
$ python3 mandelbrot.py 0.1015 -0.633 .01
```
from color import Color
from complex import Complex
from picture import Picture
import stddraw
import sys

def main():
    xc = float(sys.argv[1])
    yc = float(sys.argv[2])
    size = float(sys.argv[3])
    N = 512
    ITERATIONS = 255
    picture = Picture(N, N)
    for col in range(N):
        for row in range(N):
            x0 = xc - size / 2 + size * col / N
            y0 = yc - size / 2 + size * row / N
            z0 = Complex(x0, y0)
            gray = ITERATIONS - _mandel(z0, ITERATIONS)
            color = Color(gray, gray, gray)
            picture.set(col, N - 1 - row, color)
    stddraw.setCanvasSize(N, N)
    stddraw.picture(picture)
    stddraw.show()

def _mandel(z0, iterations):
    z = z0
    for i in range(iterations):
        if abs(z) > 2.0:
            return i
        z = z * z + z0
    return iterations

if __name__ == '__main__':
    main()
from color import Color
from complex import Complex
from picture import Picture
import stddraw
import sys

def main():
    xc = float(sys.argv[1])
    yc = float(sys.argv[2])
    size = float(sys.argv[3])
    N = 512
    ITERATIONS = 255
    picture = Picture(N, N)
    for col in range(N):
        for row in range(N):
            x0 = xc - size / 2 + size * col / N
            y0 = yc - size / 2 + size * row / N
            z0 = Complex(x0, y0)
            gray = ITERATIONS - _mandel(z0, ITERATIONS)
            color = Color(gray, gray, gray)
            picture.set(col, N - 1 - row, color)
    stddraw.setCanvasSize(N, N)
    stddraw.picture(picture)
    stddraw.show()

def _mandel(z0, iterations):
    z = z0
    for i in range(iterations):
        if abs(z) > 2.0:
            return i
        z = z * z + z0
    return iterations

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

A spatial vector is an abstract entity that has a magnitude and a direction.

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

- Addition: $x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$
- Subtraction: $x - y = (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n)$
- Scalar product: $\alpha x = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n)$
- Dot product: $x \cdot y = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$
- Magnitude: $|x| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2}$
- Direction: $x / |x| = (x_1 / |x|, x_2 / |x|, \ldots, x_n / |x|)$
Examples

A spatial vector is an abstract entity that has a magnitude and a direction.
A spatial vector is an abstract entity that has a magnitude and a direction.

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

- Addition: $x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$
- Subtraction: $x - y = (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n)$
- Scalar product: $\alpha x = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n)$
- Dot product: $x \cdot y = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$
- Magnitude: $|x| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2}$
- Direction: $x/|x| = (x_1/|x|, x_2/|x|, \ldots, x_n/|x|)$
Examples

A data type Vector for spatial vectors

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

class Vector:
    def __init__(self, a):
        self._n = len(a)
        self._coords = 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 dot(self, other):
        result = 0
        for i in range(self._n):
            result += self._coords[i] * other._coords[i]
        return 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)
```python
import math
import stdarray
import stdio

class Vector:
    def __init__(self, a):
        self._n = len(a)
        self._coords = 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 dot(self, other):
        result = 0
        for i in range(self._n):
            result += self._coords[i] * other._coords[i]
        return 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 direction(self):
    return self.scale(1.0 / abs(self))

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

def dimension(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('x - y = ' + str(x - y))
    stdio.writeln('x dot y = ' + str(x.dot(y)))
    stdio.writeln('10x = ' + str(x.scale(10.0)))
    stdio.writeln('xhat = ' + str(x.direction()))
    stdio.writeln('|x| = ' + str(abs(x)))
    stdio.writeln('ydim = ' + str(y.dimension()))

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

A data type Sketch for compactly representing the content of a document. Sketch(text, k, d) represents a new sketch s built from the string text using k-grams and dimension d. s.similarTo(t) is the similarity measure between sketches s and t (a float between 0.0 and 1.0). str(s) is the string representation of s.
A data type Sketch for compactly representing the content of a document

<table>
<thead>
<tr>
<th>Sketch</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>
```python
from vector import Vector
import stdarray
import stdio
import sys

class Sketch:
    def __init__(self, text, k, d):
        freq = stdarray.create1D(d, 0)
        for i in range(len(text) - k + 1):
            kgram = text[i:i+k]
            h = hash(kgram)
            freq[abs(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():
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    text = stdio.readAll()
    sketch = Sketch(text, k, d)
    stdio.writeln(sketch)

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

class Sketch:
    def __init__(self, text, k, d):
        freq = stdarray.create1D(d, 0)
        for i in range(len(text) - k + 1):
            kgram = text[i:i + k]
            h = hash(kgram)
            freq[abs(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():
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    text = stdio.readAll()
    sketch = Sketch(text, k, d)
    stdio.writeln(sketch)

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

Program: comparedocuments.py

- Command-line input: \texttt{k} (int), \texttt{d} (int), and \texttt{path} (str)
- Standard input: a document list
- Standard output: computes \texttt{d}-dimensional profiles based on \texttt{k}-gram frequencies for all those documents under the \texttt{path} directory, and writes a matrix of similarity measures between all pairs of documents

```
terminal
~/workspace/ipp/programs
$ cat ../data/documents.txt
constitution.txt
tomsawyer.txt
huckfinn.txt
tale.txt
prejudice.txt
actg.txt
djia.csv
$ python3 comparedocuments.py 5 10000 ../data < ../data/documents.txt
cons toms huck tale prej actg djia
cons 1.00 0.66 0.60 0.67 0.64 0.11 0.18
toms 0.66 1.00 0.93 0.92 0.88 0.15 0.23
huck 0.60 0.93 1.00 0.84 0.81 0.13 0.21
tale 0.67 0.92 0.84 1.00 0.87 0.14 0.21
prej 0.64 0.88 0.81 0.87 1.00 0.15 0.24
actg 0.11 0.15 0.13 0.14 0.15 1.00 0.12
djia 0.18 0.23 0.21 0.21 0.24 0.12 1.00
```
**Examples**

**Program:** comparedocuments.py

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```

---

```
Examples

Program: comparedocuments.py
```
Examples

Program: comparedocuments.py

- Command-line input: \( k \) (int), \( d \) (int), and \( path \) (str)
Examples

Program: comparedocuments.py

- Command-line input: $k$ (int), $d$ (int), and $path$ (str)
- Standard input: a document list

```
> python3 comparedocuments.py 5 10000 ../data < ../data/documents.txt

cons toms huck tale prej actg djia
cons 1.00 0.66 0.60 0.67 0.64 0.11 0.18
  toms 0.66 1.00 0.93 0.92 0.88 0.15 0.23
  huck 0.60 0.93 1.00 0.84 0.81 0.13 0.21
  tale 0.67 0.92 0.84 1.00 0.87 0.14 0.21
  prej 0.64 0.88 0.81 0.87 1.00 0.15 0.24
  actg 0.11 0.15 0.13 0.14 0.15 1.00 0.12
  djia 0.18 0.23 0.21 0.21 0.24 0.12 1.00
```
Examples

Program: comparedocuments.py

- Command-line input: $k$ (int), $d$ (int), and $path$ (str)
- Standard input: a document list
- Standard output: computes $d$-dimensional profiles based on $k$-gram frequencies for all those documents under the $path$ directory, and writes a matrix of similarity measures between all pairs of documents
Examples

Program: comparedocuments.py

- Command-line input: $k$ (int), $d$ (int), and $path$ (str)
- Standard input: a document list
- Standard output: computes $d$-dimensional profiles based on $k$-gram frequencies for all those documents under the $path$ directory, and writes a matrix of similarity measures between all pairs of documents

```
> ~/workspace/ipp/programs

$ cat ../data/documents.txt
corpus.txt
tomsawyer.txt
huckfinn.txt
tale.txt
prejudice.txt
actg.txt
djia.csv

$ python3 comparedocuments.py 5 10000 ../data < ../data/documents.txt

cons toms huck tale prej actg djia
cons 1.00 0.66 0.60 0.67 0.64 0.11 0.18
   toms 0.66 1.00 0.93 0.92 0.88 0.15 0.23
   huck 0.60 0.93 1.00 0.84 0.81 0.13 0.21
   tale 0.67 0.92 0.84 1.00 0.87 0.14 0.21
   prej 0.64 0.88 0.81 0.87 1.00 0.15 0.24
   actg 0.11 0.15 0.13 0.14 0.15 1.00 0.12
   djia 0.18 0.23 0.21 0.21 0.24 0.12 1.00
```
Examples

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

def main():
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    path = sys.argv[3]
    filenames = stdio.readAllStrings()
    n = len(filenames)
    sketches = stdarray.create1D(n, None)
    for i in range(n):
        inStream = InStream(path + '/' + filenames[i])
        text = inStream.readAll()
        sketches[i] = Sketch(text, k, d)
    stdio.write(' ')
    for filename in filenames:
        stdio.writef('%8.4s', filename)
        stdio.writeln()
    for i in range(n):
        stdio.writef('%8.4s', filenames[i])
        for j in range(n):
            stdio.writef('%8.2f', sketches[i].similarTo(sketches[j]))
        stdio.writeln()

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

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

def main():
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    path = sys.argv[3]
    filenames = stdio.readAllStrings()
    n = len(filenames)
    sketches = stdarray.create1D(n, None)
    for i in range(n):
        inStream = InStream(path + '/' + filenames[i])
        text = inStream.readAll()
        sketches[i] = Sketch(text, k, d)
    stdio.write(' ')
    for filename in filenames:
        stdio.writef(' %8.4s', filename)
    stdio.writeln()
    for i in range(n):
        stdio.writef(' %8.2f', sketches[i].similarTo(sketches[j]))
        stdio.writeln()

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

A data type `Counter` for counting:

`Counter(id, maxCount)`

a new counter named `id`, with maximum value `maxCount`

`c.increment()`

increment `c`, unless its value is `maxCount`

`c.tally()`

value of `c`

`c.reset()`

reset value of `c`

`c < d`

is `c` less than `d`?

`c == d`

are `c` and `d` equal?

`str(c)`

string representation of `c`
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.tally()</code></td>
<td>value of <code>c</code></td>
</tr>
<tr>
<td><code>c.reset()</code></td>
<td>reset value of <code>c</code></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 == d</code></td>
<td>are <code>c</code> and <code>d</code> equal?</td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of <code>c</code></td>
</tr>
</tbody>
</table>
import stdarray
import stdio
import stdrandom
import sys

class Counter:
    def __init__(self, id):
        self._id = id
        self._count = 0

    def increment(self):
        self._count += 1

    def tally(self):
        return self._count

    def reset(self):
        self._count = 0

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

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

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

def _main():
    n = int(sys.argv[1])
    trials = int(sys.argv[2])
    counters = stdarray.create1D(n, None)
    for i in range(n):
        counters[i] = Counter('counter ' + str(i))
    for i in range(trials):
import stdarray
import stdio
import stdrandom
import sys

class Counter:
    def __init__(self, id):
        self._id = id
        self._count = 0

    def increment(self):
        self._count += 1

    def tally(self):
        return self._count

    def reset(self):
        self._count = 0

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

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

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

def _main():
    n = int(sys.argv[1])
    trials = int(sys.argv[2])
    counters = stdarray.create1D(n, None)
    for i in range(n):
        counters[i] = Counter('counter ' + str(i))
    for i in range(trials):
counter.py

counters[stdrandom.uniformInt(0, n)].increment()

for counter in sorted(counters):
    stdio.writeln(counter)

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

```
~/workspace/ipp/programs

$ python3 counter.py 6 10000
1620 counter 0
1629 counter 3
1653 counter 2
1686 counter 1
1686 counter 4
1726 counter 5
```
Examples

```bash
$ python3 counter.py 6 10000
1620  counter  0
1629  counter  3
1653  counter  2
1686  counter  1
1686  counter  4
1726  counter  5
```
Examples

A comparable data type 

Country 

that represents a country by its name, capital, and population

\[
\text{Country(name, capital, population)}
\]

c\ is the country \(c\) given its name, capital, and population.

c < d is the country \(c\) less than country \(d\) by name?

c == d is the country \(c\) equal to country \(d\) by population?

\[\text{str}(c)\] 

string representation of \(c\)
A comparable data type `Country` that represents a country by its name, capital, and population

<table>
<thead>
<tr>
<th>Country(name, capital, population)</th>
<th>constructs a country <code>c</code> given its name, capital, and population</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c &lt; d</code></td>
<td>is the country <code>c</code> less than country <code>d</code> by name?</td>
</tr>
<tr>
<td><code>c == d</code></td>
<td>is the country <code>c</code> equal to country <code>d</code> by population?</td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of <code>c</code></td>
</tr>
</tbody>
</table>
import stdarray
import stdio

class Country:
    def __init__(self, name, capital, population):
        self._name = name
        self._capital = capital
        self._population = population

    def __lt__(self, other):
        return self._name < other._name

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

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

def _main():
    countries = stdarray.create1D(5, None)
    countries[0] = Country('United States', 'Washington, D.C.', 329334246)
    countries[1] = Country('Pakistan', 'Islamabad', 218719520)
    stdio.writeln('Unsorted:)
    for country in countries:
        stdio.writeln(country)
    stdio.writeln()
    stdio.writeln('Sorted by name:)
    for country in sorted(countries):
        stdio.writeln(country)
    stdio.writeln()
    stdio.writeln('Sorted by capital:)
    for country in sorted(countries, key=lambda country: country._capital):
import stdarray
import stdio

class Country:
    def __init__(self, name, capital, population):
        self._name = name
        self._capital = capital
        self._population = population

    def __lt__(self, other):
        return self._name < other._name

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

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

def _main():
    countries = stdarray.create1D(5, None)
    countries[0] = Country('United States', 'Washington, D.C.', 329334246)
    countries[1] = Country('Pakistan', 'Islamabad', 218719520)
    stdio.writeln('Unsorted: ')
    for country in countries:
        stdio.writeln(country)
    stdio.writeln()
    stdio.writeln('Sorted by name: ')
    for country in sorted(countries):
        stdio.writeln(country)
    stdio.writeln()
    stdio.writeln('Sorted by capital: ')
    for country in sorted(countries, key=lambda country: country._capital):
Examples

```python
stdio.writeln(country)
stdio.writeln()
stdio.writeln('Sorted by population: ')
for country in sorted(countries, key=lambda country: country._population):
    stdio.writeln(country)
stdio.writeln()
stdio.writeln('Reverse sorted by population: ')
for country in sorted(countries, key=lambda country: country._population, reverse=True):
    stdio.writeln(country)
```

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

```
country.py

    stdio.writeln(country)
stdio.writeln()
stdio.writeln('Sorted by population:')
    for country in sorted(countries, key=lambda country: country._population):
        stdio.writeln(country)
stdio.writeln()
stdio.writeln('Reverse sorted by population:')
    for country in sorted(countries, key=lambda country: country._population, reverse=True):
        stdio.writeln(country)

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

```bash
python3 country .py

Unsorted:
United States (Washington, D.C.): 329334246
Pakistan (Islamabad): 218719520
India (New Delhi): 1358989650
China (Beijing): 1401463880
Indonesia (Jakarta): 266911900

Sorted by name:
China (Beijing): 1401463880
India (New Delhi): 1358989650
Indonesia (Jakarta): 266911900
Pakistan (Islamabad): 218719520
United States (Washington, D.C.): 329334246

Sorted by capital:
China (Beijing): 1401463880
Pakistan (Islamabad): 218719520
Indonesia (Jakarta): 266911900
India (New Delhi): 1358989650
United States (Washington, D.C.): 329334246

Sorted by population:
Pakistan (Islamabad): 218719520
Indonesia (Jakarta): 266911900
United States (Washington, D.C.): 329334246
India (New Delhi): 1358989650
China (Beijing): 1401463880

Reverse sorted by population:
China (Beijing): 1401463880
India (New Delhi): 1358989650
United States (Washington, D.C.): 329334246
Indonesia (Jakarta): 266911900
Pakistan (Islamabad): 218719520
```
Examples

```bash
$ python3 country.py

Unsorted:
United States (Washington, D.C.): 329334246
Pakistan (Islamabad): 218719520
India (New Delhi): 1358989650
China (Beijing): 1401463880
Indonesia (Jakarta): 266911900

Sorted by name:
China (Beijing): 1401463880
India (New Delhi): 1358989650
Indonesia (Jakarta): 266911900
Pakistan (Islamabad): 218719520
United States (Washington, D.C.): 329334246

Sorted by capital:
China (Beijing): 1401463880
Pakistan (Islamabad): 218719520
Indonesia (Jakarta): 266911900
India (New Delhi): 1358989650
United States (Washington, D.C.): 329334246

Sorted by population:
Pakistan (Islamabad): 218719520
Indonesia (Jakarta): 266911900
United States (Washington, D.C.): 329334246
India (New Delhi): 1358989650
China (Beijing): 1401463880

Reverse sorted by population:
China (Beijing): 1401463880
India (New Delhi): 1358989650
United States (Washington, D.C.): 329334246
Indonesia (Jakarta): 266911900
Pakistan (Islamabad): 218719520
```
Examples

An iterable data type for iterating over Fibonacci sequences:

```python
FibonacciSequence(n)
```

a new object `f` for iterating over the first `n` Fibonacci numbers

```python
iter(f)
```

an iterable object `fiter`

```python
next(fiter)
```

the next number in the Fibonacci sequence `fiter`
An iterable \texttt{FibonacciSequence} data type for iterating over Fibonacci sequences

<table>
<thead>
<tr>
<th>\texttt{FibonacciSequence}</th>
<th>a new object \textit{f} for iterating over the first \textit{n} Fibonacci numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{FibonacciSequence(n)}</td>
<td>a new object \textit{f} for iterating over the first \textit{n} Fibonacci numbers</td>
</tr>
<tr>
<td>\texttt{iter(f)}</td>
<td>an iterable object \textit{fiter} on \textit{f}</td>
</tr>
<tr>
<td>\texttt{next(fiter)}</td>
<td>the next number in the Fibonacci sequence \textit{fiter}</td>
</tr>
</tbody>
</table>
import stdio
import sys

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

    def __iter__(self):
        return self

    def __next__(self):
        self._count += 1
        if self._count > self._n:
            raise StopIteration()
        if self._count <= 2:
            return 1
        temp = self._a
        self._a = self._b
        self._b += temp
        return self._b

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

if __name__ == '__main__':
    _main()
import stdio
import sys
class FibonacciSequence:
    def __init__(self, n):
        self._n = n
        self._a = 1
        self._b = 1
        self._count = 0
    def __iter__(self):
        return self
    def __next__(self):
        self._count += 1
        if self._count > self._n:
            raise StopIteration()
        if self._count <= 2:
            return 1
        temp = self._a
        self._a = self._b
        self._b += temp
        return self._b
def _main():
    n = int(sys.argv[1])
    for v in FibonacciSequence(n):
        stdio.writeln(v)
if __name__ == '__main__':
    _main()
Examples

/terminal
~/workspace/ipp/programs
$
python3 fibonaccisequence .py 10
1
1
2
3
5
8
13
21
34
55
### Examples

```
~$ /workspace/ipp/programs
```

```
$ python3 fibonacci_sequence.py 10
1
1
2
3
5
8
13
21
34
55
```
Exceptions

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.')
```

We can handle exceptions using a `try-except` block.
An exception is a disruptive event that occurs while a program is running, often to signal an error.
Exceptions

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)
Exceptions

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.')
```
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. ')
```

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

Program: errorhandling.py

- Command-line input: x (float)
- Standard output: square root of x, reporting an error if x is not specified, is not a float, or is negative

```bash
~/workspace/ipp/programs$
python3 errorhandling.py
x not specified
python3 errorhandling.py two
x must be a float
python3 errorhandling.py -2
x must be positive
python3 errorhandling.py 2
1.4142135623730951
```
Exceptions

Program: errorhandling.py

- Command-line input: \( x \) (float)
- Standard output: square root of \( x \), reporting an error if \( x \) is not specified, is not a float, or is negative

```
~/workspace/ipp/programs
$ python3 errorhandling.py
x not specified
$ python3 errorhandling.py two
x must be a float
$ python3 errorhandling.py -2
x must be positive
$ python3 errorhandling.py 2
1.4142135623730951
```
Program: errorhandling.py

- Command-line input: $x$ (float)
Program: errorhandling.py

- Command-line input: \( x \) (float)
- Standard output: square root of \( x \), reporting an error if \( x \) is not specified, is not a float, or is negative
Exceptions

Program: errorhandling.py

- Command-line input: x (float)
- Standard output: square root of x, reporting an error if x is not specified, is not a float, or is negative

```
> ~/workspace/ipp/programs

$ python3 errorhandling.py
x not specified
$ python3 errorhandling.py two
x must be a float
$ python3 errorhandling.py -2
x must be positive
$ python3 errorhandling.py 2
1.4142135623730951
```
Exceptions

```python
import math
import stdio
import sys

def main():
    try:
        x = float(sys.argv[1])
        result = _sqrt(x)
        stdio.writeln(result)
    except IndexError as e:
        stdio.writeln('x not specified')
    except ValueError as e:
        stdio.writeln('x must be a float')
    except Exception as e:
        stdio.writeln(e)
    finally:
        stdio.writeln('Done!')

def _sqrt(x):
    if x < 0:
        raise Exception('x must be positive')
    return math.sqrt(x)

if __name__ == '__main__':
    main()
```
```python
import math
import stdio
import sys

def main():
    try:
        x = float(sys.argv[1])
        result = _sqrt(x)
        stdio.writeln(result)
    except IndexError as e:
        stdio.writeln('x not specified')
    except ValueError as e:
        stdio.writeln('x must be a float')
    except Exception as e:
        stdio.writeln(e)
    finally:
        stdio.writeln('Done!')

def _sqrt(x):
    if x < 0:
        raise Exception('x must be positive')
    return math.sqrt(x)

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