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

APIs

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.
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 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)
```

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.

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.

### 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>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 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>

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)
```

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 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 \( z = r e^{i \theta} \)

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

complexpolar.py: Complex data type redux.
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
stdio.writeln(z)
if __name__ == '__main__':
    _main()
$ python complexpolar.py
-7.0 + 7.0i
```

Examples

```python
A data type Counter for counting

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

Examples

```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
    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: 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 \( x = (x_1, x_2, \ldots, x_n) \), \( y = (y_1, y_2, \ldots, y_n) \), 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:** \( x / |x| = (x_1 / |x|, x_2 / |x|, \ldots, x_n / |x|) \)

---

**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>

**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)
    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)
    print('x = ' + str(x))
    print('y = ' + str(y))
    print('x + y = ' + str(x + y))
    print('10x = ' + str(10 * x))
    print('|x| = ' + str(abs(x)))
    print('<x, y> = ' + str(x.dot(y)))
    if __name__ == '__main__':
        _main()
```

```python
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

21 / 37

A data type Sketch for compactly representing the content of a document

method description

Sketch(text, k, d) a new sketch s built from the string text using k-grams and dimension d
s.similarTo(t) similarity measure between sketches s and t (a float between 0.0 and 1.0)
str(s) string representation of s

Examples

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 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('%8.4 s', filenames[i])
        stdio.writeln()
    for i in range(len(filenames)):
        stdio.writef(' %.4 s', filenames[i])
        for j in range(len(filenames)):
            stdio.writef('%8.2 f', sketches[i].similarTo(sketches[j]))
            stdio.writeln()
if __name__ == '__main__':
    main()
Examples

```python
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()
```

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
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>
```

Examples

```
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

_fibonacci.py: Definition of Fibonacci data type.

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

Examples

```
$ python fibonacci.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
Saturn, 62
Jupiter, 67
```

Examples

```
_fibonacci.py: Definition of Fibonacci data type.
```

Examples

```
$ python fibonacci.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
Saturn, 62
Jupiter, 67
```
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)
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 subclustering, 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>
    ...
```

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
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('V Vectors have different dimensions')
```

We can handle exceptions using a try-except block:

```
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:

```
try:
    z = x / y
except ZeroDivisionError as e:
    z = e
```
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'
```