Using Data Types
Outline

1 Methods

2 Integers, Floats, Booleans, and Strings

3 Collections

4 User-Defined Data Types

5 Memory Management
Methods

A method is a function associated with a specific object (and, by extension, with the type of that object)

A method corresponds to a data-type operation

We call (or invoke) a method using a variable name, followed by the dot operator (.), followed by the method name, followed by its arguments separated by commas and enclosed in parentheses

Example (bits.py)

```python
import stdio

x, y, z = 200, 300, 600
xbits, ybits, zbits = x.bit_length(), y.bit_length(), z.bit_length()
stdio.writeln(xbits)
stdio.writeln(ybits)
stdio.writeln(zbits)

$ python bits.py
8
9
10
```
Integers, Floats, Booleans, and Strings

Methods in the built-in `int` data type

```python
>>> dir(int)
['bit_length', 'conjugate']
```

Methods in the built-in `float` data type

```python
>>> dir(float)
['as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'is_integer']
```

Methods in the built-in `bool` data type

```python
>>> dir(bool)
['bit_length', 'conjugate']
```

Methods in the built-in `str` data type

```python
>>> dir(str)
['capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```
Integers, Floats, Booleans, and Strings

potentialgene.py: Accept a DNA sequence as a command-line argument, and write True to standard output if the sequence corresponds to a potential gene, and False otherwise.

```python
import stdio
import sys

def isPotentialGene(dna):
    if (len(dna) % 3) != 0: return False
    if not dna.startswith('ATG'): return False
    for i in range(len(dna) - 3):
        if i % 3 == 0:
            if dna[i:i + 3] == 'TAA': return False
            if dna[i:i + 3] == 'TAG': return False
            if dna[i:i + 3] == 'TGA': return False
    if dna.endswith('TAA'): return True
    if dna.endswith('TAG'): return True
    if dna.endswith('TGA'): return True
    return False

def main():
    dna = sys.argv[1]
    stdio.writeln(isPotentialGene(dna))

if __name__ == '__main__':
    main()

$ python potentialgene.py ATGCGCCTGCCTGTACTAG
True
$ python potentialgene.py ATGCCTCGTCTGTACTAG
False
```
Collections

Methods in the built-in list data type

```python
>>> dir(list)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Methods in the built-in tuple data type

```python
>>> dir(tuple)
['count', 'index']
```

Methods in the built-in set data type

```python
>>> dir(set)
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

Methods in the built-in dict data type

```python
>>> dir(dict)
['clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
```
User-Defined Data Types

A data type `Charge` for charged particles

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Charge(x0, y0, q0)</code></td>
<td>a new charge $c$ centered at $(x_0, y_0)$ with charge value $q_0$</td>
</tr>
<tr>
<td><code>c.potentialAt(x, y)</code></td>
<td>electric potential of $c$ at point $(x, y)$</td>
</tr>
<tr>
<td><code>str(c)</code></td>
<td>string representation of $c$</td>
</tr>
</tbody>
</table>

To create an object of a user-defined data type, we call its constructor, using the name of the data type, followed by the constructor’s arguments.

We use a variable name to identify the object to be associated with the method we intend to call.

In any data-type implementation, it is worthwhile to include an operation that converts an object’s value to a string.

We use the following form of the `import` statement to import a data type `XYZ` defined in a file `xyz.py`:

```python
from xyz import XYZ
```
User-Defined Data Types

chargeclient.py: Accept floats \(x\) and \(y\) as command-line arguments, create two Charge objects with fixed position and electrical charge, and write to standard output the potential at \((x, y)\) due to the two charges.

```python
import stdio
import sys
from charge import Charge

def main():
    x = float(sys.argv[1])
    y = float(sys.argv[2])
    c1 = Charge(.51, .63, 21.3)
    c2 = Charge(.13, .94, 81.9)
    v1 = c1.potentialAt(x, y)
    v2 = c2.potentialAt(x, y)
    stdio.writeln('potential at (%.2f, %.2f) due to
' % (x, y))
    stdio.writeln(' ' + str(c1) + ' and ')
    stdio.writeln(' ' + str(c2) + ' is %.2e
' % (v1 + v2))

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

$ python chargeclient.py .2 .5
potential at (0.20, 0.50) due to
    21.3 at (0.51, 0.63) and
    81.9 at (0.13, 0.94)
is 2.22e+12

$ python chargeclient.py .51 .94
potential at (0.51, 0.94) due to
    21.3 at (0.51, 0.63) and
    81.9 at (0.13, 0.94)
is 2.56e+12
### User-Defined Data Types

A data type `Color` for representing color values

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color(r, g, b)</td>
<td>a new color <code>c</code> with red, green, and blue components <code>r</code>, <code>g</code>, and <code>b</code>, all integers between 0 and 255</td>
</tr>
<tr>
<td>c.getRed()</td>
<td>the red component of <code>c</code></td>
</tr>
<tr>
<td>c.getGreen()</td>
<td>the green component of <code>c</code></td>
</tr>
<tr>
<td>c.getBlue()</td>
<td>the blue component of <code>c</code></td>
</tr>
<tr>
<td>str(c)</td>
<td>string representation of <code>c</code></td>
</tr>
</tbody>
</table>
User-Defined Data Types

alberssquares.py: Accept integers $r_1, g_1, b_1, r_2, g_2,$ and $b_2$ as command-line arguments, and draw to standard draw Albers squares using colors $(r_1, g_1, b_1)$ and $(r_2, g_2, b_2)$.

```python
import stddraw
import sys
from color import Color

def main():
    r1 = int(sys.argv[1])
    g1 = int(sys.argv[2])
    b1 = int(sys.argv[3])
    c1 = Color(r1, g1, b1)
    r2 = int(sys.argv[4])
    g2 = int(sys.argv[5])
    b2 = int(sys.argv[6])
    c2 = Color(r2, g2, b2)
    stddraw.setCanvasSize(512, 256)
    stddraw.setYscale(.25, .75)
    stddraw.setPenColor(c1)
    stddraw.filledSquare(.25, .5, .2)
    stddraw.setPenColor(c2)
    stddraw.filledSquare(.25, .5, .1)
    stddraw.setPenColor(c2)
    stddraw.filledSquare(.75, .5, .2)
    stddraw.setPenColor(c1)
    stddraw.filledSquare(.75, .5, .1)
    stddraw.show()

if __name__ == '__main__':
    main()
```
User-Defined Data Types

$ python alberssquares.py 9 90 166 100 100 100

$ python alberssquares.py 0 174 239 147 149 252

$ python alberssquares.py 110 110 110 145 160 156
luminance.py: Accept six RGB values defining two colors as command-line arguments, and write `True` to standard output if the two colors are compatible, and `False` otherwise.

```python
import stdio
import sys
from color import Color

def luminance(c):
    red, green, blue = c.getRed(), c.getGreen(), c.getBlue()
    return (.299 * red) + (.587 * green) + (.114 * blue)

def toGray(c):
    y = int(round(luminance(c)))
    return Color(y, y, y)

def areCompatible(c1, c2):
    return abs(luminance(c1) - luminance(c2)) >= 128.0

def main():
    r1, g1, b1 = int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3])
    r2, g2, b2 = int(sys.argv[4]), int(sys.argv[5]), int(sys.argv[6])
    c1, c2 = Color(r1, g1, b1), Color(r2, g2, b2)
    stdio.writeln(areCompatible(c1, c2))

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

$ python luminance.py 232 232 232 0 0 0
True
$ python luminance.py 9 90 166 232 232 232
True
$ python luminance.py 9 90 166 0 0 0
False
A data type `Picture` for representing digital images

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Picture(w, h)</code></td>
<td>a new (w)-by-(h) picture (pic)</td>
</tr>
<tr>
<td><code>Picture(filename)</code></td>
<td>a new picture (pic) initialized from (filename)</td>
</tr>
<tr>
<td><code>pic.save(filename)</code></td>
<td>save (pic) to (filename)</td>
</tr>
<tr>
<td><code>pic.width()</code></td>
<td>the width of (pic)</td>
</tr>
<tr>
<td><code>pic.height()</code></td>
<td>the height of (pic)</td>
</tr>
<tr>
<td><code>pic.get(col, row)</code></td>
<td>the color of pixel ((col, row)) in (pic)</td>
</tr>
<tr>
<td><code>pic.set(col, row, c)</code></td>
<td>set the color of pixel ((col, row)) in (pic) to (c)</td>
</tr>
</tbody>
</table>
grayscale.py: Accept the name of a JPG or PNG file as a command-line argument, read an image from the file, and draw a gray scale version of that image to standard draw.

```python
import luminance
import stddraw
import sys
from picture import Picture

def main():
    pic = Picture(sys.argv[1])
    for col in range(pic.width()):
        for row in range(pic.height()):
            pixel = pic.get(col, row)
            gray = luminance.toGray(pixel)
            pic.set(col, row, gray)
    stddraw.setCanvasSize(pic.width(), pic.height())
    stddraw.picture(pic)
    stddraw.show()

if __name__ == '__main__':
    main()

$ python grayscale.py mandrill.jpg
```
User-Defined Data Types

def main():
    fileName = sys.argv[1]
    w = int(sys.argv[2])
    h = int(sys.argv[3])
    source = Picture(fileName)
    target = Picture(w, h)
    for tCol in range(w):
        for tRow in range(h):
            sCol = tCol * source.width() // w
            sRow = tRow * source.height() // h
            target.set(tCol, tRow, source.get(sCol, sRow))
    stddraw.setCanvasSize(w, h)
    stddraw.picture(target)
    stddraw.show()
User-Defined Data Types

$ python scale.py mandrill.jpg 200 200

$ python scale.py mandrill.jpg 200 100

$ python scale.py mandrill.jpg 100 200
User-Defined Data Types

fade.py: Accept strings sourceFile and targetFile and integer n as command-line arguments, and then, over the course of n frames, gradually replace the image from sourceFile with the image from targetFile, and display to standard draw each intermediate image.

```python
import stddraw
import sys
from color import Color
from picture import Picture

def blend(c1, c2, alpha):
    r = (1 - alpha) * c1.getRed() + alpha * c2.getRed()
    g = (1 - alpha) * c1.getGreen() + alpha * c2.getGreen()
    b = (1 - alpha) * c1.getBlue() + alpha * c2.getBlue()
    return Color(int(r), int(g), int(b))

def main():
    sourceFile = sys.argv[1]
    targetFile = sys.argv[2]
    n = int(sys.argv[3])
    source = Picture(sourceFile)
    target = Picture(targetFile)
    width = source.width()
    height = source.height()
    stddraw.setCanvasSize(width, height)
    pic = Picture(width, height)
```
User-Defined Data Types

```python
for t in range(n + 1):
    for col in range(width):
        for row in range(height):
            c0 = source.get(col, row)
            cn = target.get(col, row)
            alpha = float(t) / float(n)
            c = blend(c0, cn, alpha)
            pic.set(col, row, c)
            stddraw.picture(pic)
            stddraw.show(1000.0)
            stddraw.show()

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

$ python fade.py mandrill.jpg darwin.jpg 5
potential.py: Read values from standard input to create a list of charged particles, set each pixel color in an image to a grayscale value proportional to the potential due to the particles at corresponding points, and draw the resulting image to standard draw.

```python
import stdarray
import stddraw
import stdio
from charge import Charge
from color import Color
from picture import Picture

def main():
    MAX_GRAY_SCALE = 255
    WIDTH = 512
    HEIGHT = 512
    n = stdio.readInt()
    charges = stdarray.create1D(n)
    for i in range(n):
        x0 = stdio.readFloat()
        y0 = stdio.readFloat()
        q0 = stdio.readFloat()
        charges[i] = Charge(x0, y0, q0)
    pic = Picture(WIDTH, HEIGHT)
    for col in range(pic.width()):
        for row in range(pic.height()):
            x = 1.0 * col / pic.width()
            y = 1.0 * row / pic.height()
            v = 0.0
            for i in range(n):
                v += charges[i].potentialAt(x, y)
```
User-Defined Data Types

```python
v = (MAX_GRAY_SCALE / 2.0) + (v / 2.0e10)
if v < 0:
    grayScale = 0
elif v > MAX_GRAY_SCALE:
    grayScale = MAX_GRAY_SCALE
else:
    grayScale = int(v)

color = Color(grayScale, grayScale, grayScale)
pic.set(col, pic.height() - 1 - row, color)
stddraw.setCanvasSize(pic.width(), pic.height())
stddraw.picture(pic)
stddraw.show()
```

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

$ more charges.txt
9
0.51 0.63 -100
0.50 0.50 40
0.50 0.72 10
0.33 0.33 5
0.20 0.20 -10
0.70 0.70 10
0.82 0.72 20
0.85 0.23 30
0.90 0.12 -50

$ python potential.py < charges.txt
User-Defined Data Types

A data type \texttt{InStream} that supports reading numbers and text from files and websites as well as the standard input

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{InStream(filename)}</td>
<td>a new input stream \textit{in}, initialized from \textit{filename} (defaults to standard input)</td>
</tr>
<tr>
<td>in.isEmpty()</td>
<td>is \textit{in} empty?</td>
</tr>
<tr>
<td>in.readInt()</td>
<td>read a token from \textit{in}, and return it as an integer</td>
</tr>
<tr>
<td>in.readFloat()</td>
<td>read a token from \textit{in}, and return it as a float</td>
</tr>
<tr>
<td>in.readBool()</td>
<td>read a token from \textit{in}, and return it as a boolean</td>
</tr>
<tr>
<td>in.readString()</td>
<td>read a token from \textit{in}, and return it as a string</td>
</tr>
</tbody>
</table>

A data type \texttt{OutStream} that supports writing strings to a variety of output streams, including standard output and files

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{OutStream(filename)}</td>
<td>a new output stream \textit{out} that will write to \textit{filename} (defaults to standard output)</td>
</tr>
<tr>
<td>out.write(x)</td>
<td>write \textit{x} to \textit{out}</td>
</tr>
<tr>
<td>out.writeln(x)</td>
<td>write \textit{x} to \textit{out}, followed by a newline</td>
</tr>
<tr>
<td>out.writef(fmt, arg1, ...)</td>
<td>write the arguments \textit{arg} \textsc{1}, \ldots \text{ to } \textit{out} as specified by the format string \textit{fmt}</td>
</tr>
</tbody>
</table>
import sys
from instream import InStream
from outstream import OutStream

def main():
    inFilenames = sys.argv[1:len(sys.argv) - 1]
    outFilename = sys.argv[len(sys.argv) - 1]
    outstream = OutStream(outFilename)
    for filename in inFilenames:
        instream = InStream(filename)
        s = instream.readAll()
        outstream.write(s)

if __name__ == '__main__':
    main()

$ more in1.txt
This is
$ more in2.txt
a tiny
test.

$ python cat.py in1.txt in2.txt out.txt

$ more out.txt
This is
a tiny
test.
import stdio
import sys
from instream import InStream

def _readHTML(stockSymbol):
    WEBSITE = 'http://finance.yahoo.com/q?s=
    page = InStream(WEBSITE + stockSymbol)
    html = page.readAll()
    return html

def priceOf(stockSymbol):
    html = _readHTML(stockSymbol)
    trade = html.find('yfs_l84', 0)
    beg = html.find('>', trade)
    end = html.find('</span>', beg)
    price = html[beg+1:end]
    price = price.replace(',', '')
    return float(price)

def main():
    stockSymbol = sys.argv[1]
    price = priceOf(stockSymbol)
    stdio.writeln('%.2f
', price)

if __name__ == '__main__':
    main()

$ python stockquote.py goog
733.76
User-Defined Data Types

split.py: Accept string \texttt{fileName} and integer \texttt{fieldCount} as command-line arguments, split the file whose name is \texttt{fileName.csv}, by field, into \texttt{fieldCount} files named \texttt{fileName0.txt}, \texttt{fileName1.txt}, etc.

```python
import stdarray
import sys
from instream import InStream
from outstream import OutStream

def main():
    DELIM = ','
    fileName = sys.argv[1]
    fieldCount = int(sys.argv[2])
    inStream = InStream(fileName + '.csv')
    outStreams = stdarray.create1D(fieldCount)
    for i in range(fieldCount):
        outStreams[i] = OutStream(fileName + str(i) + '.txt')
    while inStream.hasNextLine():
        line = inStream.readLine()
        fields = line.split(DELIM)
        for i in range(fieldCount):
            outStreams[i].writeln(fields[i])

if __name__ == '__main__':
    main()
```
User-Defined Data Types

$ more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
...

$ python split.py ip 2

$ more ip0.txt
www.princeton.edu
www.cs.princeton.edu
www.math.princeton.edu
...
$ more ip1.txt
128.112.128.15
128.112.136.35
128.112.18.11
...
Memory Management

Each time we create an object, Python reserves computer memory for it.

An orphaned object is an object that cannot be referenced by a program.

Python implements garbage collection — managing memory by keeping track of orphaned objects and returning the memory they use to a pool of free memory.