Collections

Lists

A collection is a way to organize data that we wish to process with a computer program.

A list (aka array) is a collection that stores a sequence of (references to) objects.

The simplest way to create a list (an object of the built-in sequence type `list`) in Python is to place comma-separated values between matching square brackets.

For example, the following code creates a list `suits` with four strings and list `x` with three floats:

```python
suits = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
x = [0.30, 0.60, 0.10]
```

After creating a list, we can refer to any individual object by specifying the list name followed by an integer index within square brackets.

For example, if we have two lists of floats `x` and `y` whose length is given by a variable `n`, we can calculate their dot product as follows:

```python
total = 0.0
for i in range(n):
    total += x[i] * y[i]
```

We refer to the first element of an `n`-element list `a` as `a[0]`, the second element as `a[1]`, and so on; the last (nth) element is referred to as `a[n - 1]`.

We can access the length of a list `a` using the built-in function `len()`.

We can use the `+=` operator to append elements to a list.

For example, the following code creates a list `a` with `n` floats, with each element initialized to `0.0`:

```python
a = []
for i in range(n):
    a += [0.0]
```

Lists are fundamental data structures in that they have a direct correspondence with memory.

```
<table>
<thead>
<tr>
<th>i</th>
<th>x[i]</th>
<th>y[i]</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>6.0</td>
<td>18.0</td>
</tr>
</tbody>
</table>
```

Outline

1. Lists
2. Tuples
3. Sets
4. Dictionaries
5. Comprehensions
6. Looping Techniques
Lists

Lists are mutable objects because we can change their values.

For example, the following code reverses the order of elements in a list `a`:

```python
n = len(a)
for i in range(n // 2):
temp = a[i]
a[i] = a[n - 1 - i]
a[n - 1 - i] = temp
```

Variable trace when `a = [1, 2, 3, 4, 5]`:

<table>
<thead>
<tr>
<th><code>i</code></th>
<th><code>n - 1 - i</code></th>
<th><code>a</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>[5, 2, 3, 4, 1]</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>[5, 4, 3, 2, 1]</td>
</tr>
</tbody>
</table>

One of the most basic operations on a list is to iterate over all its elements.

For example, the following code computes the average of a list of floats:

```python
total = 0.0
for i in range(len(a)):
total += a[i]
average = total / len(a)
```

Variable trace when `a = [1.0, 2.0, 3.0]`:

<table>
<thead>
<tr>
<th><code>i</code></th>
<th><code>a[i]</code></th>
<th><code>total</code></th>
<th><code>average</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
<td>3.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>6.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Python has several built-in functions that take lists as arguments.

For example, given a list `a`:

- `len(a)` returns the number of elements in the list.
- `sum(a)` returns the sum of the elements in the list.
- `min(a)` returns the minimum element in the list.
- `max(a)` returns the maximum element in the list.
- ...

We can write a list by passing it as an argument to `stdio.write()` or `stdio.writeln()`, or we can use a `for` statement to write each element individually.

Example: Code for representing and processing playing cards.

Represent suits and ranks:

```python
SUITS = ['Clubs ', 'Diamonds ', 'Hearts ', 'Spades ']
RANKS = ['2', '3', '4', '5', '6', '7', '8', '9', '10 ',
         'Jack ', 'Queen ', 'King ', 'Ace ']
```

Write a random card name:

```python
rank = random.randrange(0, len(RANKS))
suit = random.randrange(0, len(SUITS))
stdio.writeln(RANKS[rank] + ' of ' + SUITS[suit])
```

Create a deck:

```python
deck = []
for rank in RANKS:
    for suit in SUITS:
        card = rank + ' of ' + suit
        deck += [card]
```

Shuffle the deck:

```python
n = len(deck)
for i in range(n):
r = random.randrange(i, n)
temp = deck[r]
deck[r] = deck[i]
deck[i] = temp
```
sample.py: Accept integers m and n as command-line arguments. Write to standard output a random sample of m integers in the range $0 \ldots n - 1$ (no duplicates).

```python
import random
import stdarray
import stdio
import sys
m = int(sys.argv[1])
n = int(sys.argv[2])
perm = stdarray.create1D(n, 0)
for i in range(n):
    perm[i] = i
for i in range(m):
    r = random.randrange(i, n)
temp = perm[r]
perm[r] = perm[i]
perm[i] = temp
for i in range(m):
    stdio.write(str(perm[i]) + ' ')
stdio.writeln()
```

$ python3 sample.py 6 16
9 6 0 8 5 15

$ python3 sample.py 10 1000
389 22 385 925 611 485 866 978 212 298

$ python3 sample.py 20 20
7 18 2 16 4 10 14 0 3 13 17 8 5 1 11 6 9 12 19 15

couponcollector.py: Accept integer n as a command-line argument. Write to standard output the number of coupons you collect before obtaining one of each of n types.

```python
import random
import stdarray
import stdio
import sys
n = int(sys.argv[1])
count = 0
collectedCount = 0
isCollected = stdarray.create1D(n, False)
while collectedCount < n:
    value = random.randrange(0, n)
count += 1
    if not isCollected[value]:
        collectedCount += 1
        isCollected[value] = True
stdio.writeln(count)
```

$ python3 couponcollector.py 1000
5821

$ python3 couponcollector.py 1000
8155

$ python3 couponcollector.py 1000000
13988284

primesieve.py: Accept integer n as a command-line argument. Write to standard output the number of primes less than or equal to n.

```python
import stdarray
import stdio
import sys
n = int(sys.argv[1])
isPrime = stdarray.create1D(n + 1, True)
for i in range(2, n):
    if isPrime[i]:
        for j in range(i * i, n + 1, i):
            isPrime[j] = False
    count += 1
stdio.writeln(count)
```

$ python3 primesieve.py 10
4

$ python3 primesieve.py 1000
168

$ python3 primesieve.py 100000
9592

$ python3 primesieve.py 10000000
664579

In many applications, a convenient way to store information is to use a table of numbers organized in a rectangular table and refer to rows and columns in the table. The simplest way to create a two-dimensional list in Python is to place comma-separated one-dimensional lists between matching square brackets.

For example, the following code creates an $m$-by-$n$ list $a$ of floats, with all elements initialized to 0.0:

```python
a = []
for i in range(m):
    row = [0.0] * n
    a += [row]
```
Lists

When a is a two-dimensional list, the syntax a[i] refers to its i\textsuperscript{th} row, which is a one-dimensional list; The syntax a[i][j] refers to the object at row i and column j.

For example, the following code adds two n-by-n matrices a and b:

```python
c = stdarray.create2D(n, n, 0.0)
for i in range(n):
    for j in range(n):
        c[i][j] = a[i][j] + b[i][j]
```

Variable trace when a = \([1.0, 2.0], [3.0, 4.0]\) and b = \([2.0, 3.0], [4.0, 5.0]\):

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>a[i][j]</th>
<th>b[i][j]</th>
<th>c[i][j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3.0</td>
<td>4.0</td>
<td>7.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4.0</td>
<td>5.0</td>
<td>9.0</td>
</tr>
</tbody>
</table>

selfavoid.py: Accept integers n and trials as command-line arguments. Do trials random self-avoiding walks in an n-by-n lattice. Write to standard output the percentage of dead ends encountered.

```python
import random
import stdarray
importstdio
import sys
n = int(sys.argv[1])
trials = int(sys.argv[2])
deadEnds = 0
for t in range(trials):
    a = stdarray.create2D(n, n, False)
x = n // 2
y = n // 2
while (x > 0) and (x < n - 1) and
    (y > 0) and (y < n - 1):
    a[x][y] = True
    if a[x - 1][y] and a[x + 1][y] and
        a[x][y - 1] and a[x][y + 1]:
        deadEnds += 1
        break
    r = random.randrange(1, 5)
    if (r == 1) and (not a[x + 1][y]):
        x += 1
    elif (r == 2) and (not a[x - 1][y]):
        x -= 1
    elif (r == 3) and (not a[x][y + 1]):
        y += 1
    elif (r == 4) and (not a[x][y - 1]):
        y -= 1
stdio.writeln(str(100 * deadEnds // trials) + '
% dead ends')
```

A list with rows of nonuniform length is known as a ragged list.

For example, the following code writes the contents of a ragged list:

```python
for i in range(len(a)):
    for j in range(len(a[i])):
        stdio.write(a[i][j])
    stdio.write(' ')
stdio.writeln()```

So, the ragged list a = \([[[1, 1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]\) (representing Pascal’s triangle of order 4) is written out as:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

The same notation extends to allow us to compose code using lists that have any number of dimensions.

For example, using lists of lists of lists, we can create a three-dimensional list a, and then refer to an individual element of a as a[i][j][k].

$ python3 selfavoid.py 5 1000
0% dead ends
$ python3 selfavoid.py 20 1000
30% dead ends
$ python3 selfavoid.py 40 1000
78% dead ends
$ python3 selfavoid.py 80 1000
98% dead ends

15 / 20

13 / 22

15 / 20

16 / 22
**Tuples**

A tuple (an object of the built-in sequence type `tuple`) consists of a number of values separated by commas.

```python
>>> t = 12345, 54321, 'hello!
```

- **Accessing elements**:
  ```python
  >>> t[0]
  12345
  >>> t
  (12345, 54321, 'hello!')
  ```

- **Tuples may be nested**:
  ```python
  >>> u = t, (1, 2, 3, 4, 5)
  >>> u
  ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
  ```

- **Tuples are immutable, but they can contain mutable objects**:
  ```python
  >>> v = ([1, 2, 3], [3, 2, 1])
  >>> v
  ([1, 2, 3], [3, 2, 1])
  ```

**Empty and singleton sequences**

```python
>>> empty = ()
>>> len(empty)
0
>>> singleton = 'hello,
>>> len(singleton)
1
```

**Sequence unpacking**

```python
>>> x, y, z = t
```

**Sets**

A set (an object of the built-in sequence type `set`) is an unordered collection with no duplicate elements.

```python
>>> basket = [' apple ', 'orange ', 'apple ', 'pear ', 'orange ', 'banana ']
>>> fruit = set(basket)
>>> fruit
{'orange', 'pear', 'apple', 'banana'}
```

- **Fast membership testing**
  ```python
  >>> 'orange' in fruit
  True
  >>> 'crabgrass' in fruit
  False
  ```

**Set operations**

```python
>>> a = set(' abracadabra ')  
>>> b = set(' alacazam ')  
>>> a  
set({'a', 'r', 'b', 'c', 'd'})  
>>> a - b  
set({'r', 'd', 'b'})  
>>> a | b  
set({'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'})  
>>> a & b  
set({'a', 'c'})  
>>> a ^ b  
set({'r', 'd', 'b', 'm', 'z', 'l'})
```

**Comprehensions**

Comprehensions provide a concise way to create collections.

**List comprehensions**

```python
>>> squares = [x ** 2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```python
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

**Nested list comprehensions**

```python
>>> matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

**Set comprehensions**

```python
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```
When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function:

```python
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...    stdio.writeln(str(i) + ' ' + v)
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function:

```python
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...    stdio.writeln('What is your ' + q + '? It is ' + a)
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function:

```python
>>> for i in reversed(range(1, 10, 2)):
...    stdio.writeln(i)
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered:

```python
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...    stdio.writeln(f)
apple
banana
orange
pear
```