Collections
Outline

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

```
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.

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

**Variable trace when** \(x = [1.0, 2.0, 3.0]\), \(y = [4.0, 5.0, 6.0]\), and \(n = 3\)

<table>
<thead>
<tr>
<th>i</th>
<th>(x[i] \times y[i])</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0 * 4.0 = 4.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>2.0 * 5.0 = 10.0</td>
<td>14.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0 * 6.0 = 18.0</td>
<td>32.0</td>
</tr>
</tbody>
</table>
Lists

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 (\( n \)th) element is referred to as \( a[n-1] \).

We can access the length of a list \( a \) using the built-in function \( \text{len()} \).

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 \( 0.0 \).

\[
a = []
\]

\[
\text{for } i \text{ in } \text{range}(n):
    a += [0.0]
\]

Lists are fundamental data structures in that they have a direct correspondence with memory.
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>i</th>
<th>n - 1 - i</th>
<th>a</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>i</th>
<th>a[i]</th>
<th>total</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<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>
<tr>
<td></td>
<td></td>
<td>6.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
and the following code does the same without referring to the indices explicitly

```python
total = 0.0
for v in a:
    total += v
average = total / len(a)
```

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

```
v   total   average
-------  -------
  0.0    0.0
  1.0    1.0
  2.0    3.0
  3.0    6.0
  6.0    2.0
```

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
Lists

Aliasing refers to the situation where two variables refer to the same object

For example, after the assignment statements

\[
x = [.30, .60, .10]
y = x
x[1] = .99
\]

the value of \(y[1]\) is also .99

One way to make a copy \(y\) of a given list \(x\) is to iterate though \(x\) to build \(y\)

\[
y = []
for v in x:
    y += [v]
\]

Alternatively, the expression \(a[i:j]\), called slicing, evaluates to a new list whose elements are \(a[i], \ldots, a[j - 1]\); the default value for \(i\) is 0 and the default value for \(j\) is \(\text{len}(a)\), so \(y = x[:]\) is equivalent to the code above

The \texttt{stdarray} module from the authors of the IPP text defines functions for processing lists

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{stdarray.create1D(n, val)}</td>
<td>list of length (n), each element initialized to (val)</td>
</tr>
<tr>
<td>\texttt{stdarray.create2D(m, n, val)}</td>
<td>(m)-by-(n) list, each element initialized to (val)</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

A string is a list of characters and hence can be manipulated like a list

The familiar \texttt{sys.argv} object is a list of string objects
Lists

Example: Code for representing and processing playing cards.

Represent suits and ranks

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

Write a random card name

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

Create a deck

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

Shuffle the deck

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

9 / 22
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
```
**Lists**

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(2, n // i + 1):
            isPrime[i * j] = False

count = 0
for i in range(2, n + 1):
    if isPrime[i]:
        count += 1
stdio.writeln(count)
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
```

$ python3 primesieve.py 10
4

$ python3 primesieve.py 1000
168

$ python3 primesieve.py 100000
9592

$ python3 primesieve.py 1000000
664579
Lists
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, this matrix of integers having two rows and three columns

\[
\begin{bmatrix}
18 & 19 & 20 \\
21 & 22 & 23
\end{bmatrix}
\]

can be represented in Python using this list of lists

\[
a = \begin{bmatrix}
[18, 19, 20], 
[21, 22, 23]
\end{bmatrix}
\]

Python represents an \(m\)-by-\(n\) list \(a\) as a list that contains \(m\) objects, each of which is a list that contains \(n\) objects; so \(m = \text{len}(a)\) and \(n = \text{len}(a[0])\), assuming \(a\) is non-ragged, ie, has \(n\) elements in each row.

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

\[
a = []
\text{for i in range}(m):
    \text{row} = [0.0] * n
\text{a} += [\text{row}]
\]
When \( a \) is a two-dimensional list, the syntax \( a[i] \) refers to its \( i \)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.

import random
import stdarray
import stdio
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')
Lists

$ python3 selfavoid.py 5 1000
0% dead ends

$ python3 selfavoid.py 20 1000
30% dead ends

$ python3 selfavoid.py 40 1000
75% dead ends

$ python3 selfavoid.py 80 1000
98% dead ends
Lists

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
def write_ragged_list(a):
    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]`. 
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!

>>> 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 = ()
>>> singleton = 'hello',

>>> len empty
0
>>> 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
set(['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 # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b # letters in both a and b
set(['a', 'c'])
>>> a ^ b # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```
A dictionary (an object of the built-in mapping type `dict`) is an unordered set of key-value pairs, with the requirement that the keys are unique.

```python
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098, 'sape': 4139}
>>> 'guido' in tel
True
```

Dictionaries can be built directly from sequences of key-value pairs.

```python
>>> dict([( 'sape ', 4139), ( 'guido ', 4127), ( 'jack ', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```
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]
>>> [(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
set(['r', 'd'])
```

Dictionary comprehensions

```python
>>> {x: x ** 2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```
Looping Techniques

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.
```
Looping Techniques

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
```