Collection Data Types

Outline

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

Lists

Lists

A list (object of type `list`) is an ordered collection of objects

Lists

A list (object of type `list`) is an ordered collection of objects

Creating a list

```
<name> = [<expression>, <expression>, ..., <expression>]
```

Lists

A list (object of type `list`) is an ordered collection of objects

Creating a list

```
<name> = [<expression>, <expression>, ..., <expression>]
```

Example

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

Lists

Lists

Appending to a list

```
<name> += [<expression>]
```

Lists

Appending to a list

```
<name> += [<expression>]
```

Example (creating a list a with n zeros)

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

Lists

Appending to a list

```
<name> += [<expression>]
```

Example (creating a list `a` with `n` zeros)

```
1  a = []
2  for i in range(n):
3      a += [0.0]
```

Variable trace (`n = 3`)

line #	a	i
1	[]	
2	[]	0
3	[0.0]	0
2	[0.0]	1
3	[0.0, 0.0]	1
2	[0.0, 0.0]	2
3	[0.0, 0.0, 0.0]	2

Lists

Lists

The number of objects in a list `<name>` is obtained as `len(<name>)`

Lists

The number of objects in a list `<name>` is obtained as `len(<name>)`

The `i`th object in a list `<name>` is referred to as `<name>[i]`, where `0 <= i < len(<name>)`

Lists

The number of objects in a list `<name>` is obtained as `len(<name>)`

The `i`th object in a list `<name>` is referred to as `<name>[i]`, where `0 <= i < len(<name>)`

Example (computing the dot product of lists `x` and `y`)

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

Lists

The number of objects in a list `<name>` is obtained as `len(<name>)`

The `i`th object in a list `<name>` is referred to as `<name>[i]`, where `0 <= i < len(<name>)`

Example (computing the dot product of lists `x` and `y`)

```
1  total = 0.0
2  for i in range(len(x)):
3      total += x[i] * y[i]
```

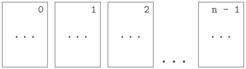
Variable trace (`x = [1.0, 2.0, 3.0]`, `y = [4.0, 5.0, 6.0]`)

line #	total	i
1	0.0	
2	0.0	0
3	4.0	0
2	4.0	1
3	14.0	1
2	14.0	2
3	32.0	2

Lists

Lists

Memory model for a list `<name>` with `n` objects



0 ... 1 ... 2 ... ... n - 1 ...

Lists

Lists

Lists are mutable

Lists

Lists are mutable

Example (reversing 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
```

Lists

Lists are mutable

Example (reversing a list `a`)

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

Variable trace (`a = [1, 2, 3, 4, 5]`)

line #	a	n	i
1	[1, 2, 3, 4, 5]	5	
2	[1, 2, 3, 4, 5]	5	0
5	[5, 2, 3, 4, 1]	5	0
2	[5, 2, 3, 4, 1]	5	1
5	[5, 4, 3, 2, 1]	5	1

Lists

Lists

Lists can be iterated by index

Lists

Lists can be iterated by index

Example (averaging the numbers in a list `a`)

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

Lists

Lists can be iterated by index

Example (averaging the numbers in a list `a`)

```
1  total = 0.0
2  for i in range(len(a)):
3      total += a[i]
4  average = total / len(a)
```

Variable trace (`a = [2.0, 4.0, 6.0]`)

line #	total	i	average
1	0.0		
2	0.0	0	
3	2.0	0	
2	2.0	1	
3	6.0	1	
2	6.0	2	
3	12.0	2	
4	12.0		4.0

Lists

Lists

Lists can also be iterated by value

Lists

Lists can also be iterated by value

Example (averaging the numbers in a list `a`)

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

Lists

Lists can also be iterated by value

Example (averaging the numbers in a list `a`)

```
1  total = 0.0
2  for v in a:
3      total += v
4  average = total / len(a)
```

Variable trace (`a = [2.0, 4.0, 6.0]`)

line #	total	v	average
1	0.0		
2	0.0	2.0	
3	2.0	2.0	
2	2.0	4.0	
3	6.0	4.0	
2	6.0	6.0	
3	12.0	6.0	
4	12.0		4.0

Lists

Lists

Python has several built-in functions that operate on lists

Lists

Python has several built-in functions that operate on lists

For example, given a list `a`:

Lists

Python has several built-in functions that operate on lists

For example, given a list `a`:

- `len(a)` returns the number of elements in the list

Lists

Python has several built-in functions that operate on lists

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

Lists

Python has several built-in functions that operate on lists

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

Lists

Python has several built-in functions that operate on lists

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

Lists

Python has several built-in functions that operate on lists

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

The `stdarray` library provides functions for creating lists

☰ `stdarray`	
`create1D(n, value = None)`	creates and returns a 1D list of size `n`, with each element initialized to `value`
`create2D(m, n, value = None)`	creates and returns a 2D list of size `m x n`, with each element initialized to `value`

Lists

Lists

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

Lists

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

Example

```
x = [1, 3, 7]
y = x
x[1] = 42
stdio.writeln(x)
stdio.writeln(y)
```

```
[1, 42, 7]
[1, 42, 7]
```

Lists

Lists

Creating a list `y` as a copy (not an alias) of `x`, using a loop

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

Lists

Creating a list `y` as a copy (not an alias) of `x`, using a loop

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

Creating a list `y` as a copy (not an alias) of `x`, using slicing

```
y = x[:]
```

Lists

Creating a list `y` as a copy (not an alias) of `x`, using a loop

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

Creating a list `y` as a copy (not an alias) of `x`, using slicing

```
y = x[:]
```

In general, `x[i:j]` returns a sublist `[x[i], ..., x[j - 1]]`, with `i = 0` and `j = len(x)` if either is unspecified

Lists

Lists

Example (playing cards)

Lists

Example (playing cards)

```
# Represent ranks and suits.
RANKS = [’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’10’, ’Jack’, ’Queen’, ’King’, ’Ace’]
SUITS = [’Clubs’, ’Diamonds’, ’Hearts’, ’Spades’]
```

Lists

Example (playing cards)

```
# Represent ranks and suits.
RANKS = [’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’10’, ’Jack’, ’Queen’, ’King’, ’Ace’]
SUITS = [’Clubs’, ’Diamonds’, ’Hearts’, ’Spades’]
```

```
# Create a deck.
deck = []
for rank in RANKS:
    for suit in SUITS:
        card = rank + ’ of ’ + suit
        deck += [card]
```

Lists

Example (playing cards)

```
# Represent ranks and suits.
RANKS = [’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’10’, ’Jack’, ’Queen’, ’King’, ’Ace’]
SUITS = [’Clubs’, ’Diamonds’, ’Hearts’, ’Spades’]
```

```
# Create a deck.
deck = []
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 = stdrandom.uniformInt(i, n)
    temp = deck[r]
    deck[r] = deck[i]
    deck[i] = temp
```

Lists

Example (playing cards)

```
# Represent ranks and suits.
RANKS = [’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’10’, ’Jack’, ’Queen’, ’King’, ’Ace’]
SUITS = [’Clubs’, ’Diamonds’, ’Hearts’, ’Spades’]
```

```
# Create a deck.
deck = []
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 = stdrandom.uniformInt(i, n)
    temp = deck[r]
    deck[r] = deck[i]
    deck[i] = temp
```

```
# Draw a random card from the deck and write it to standard output.
rank = stdrandom.uniformInt(0, len(RANKS))
suit = stdrandom.uniformInt(0, len(SUITS))
stdio.writeln(RANKS[rank] + ’ of ’ + SUITS[suit])
```

Lists

Lists

Program: `sample.py`

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/ipp/programs
$ _
```

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

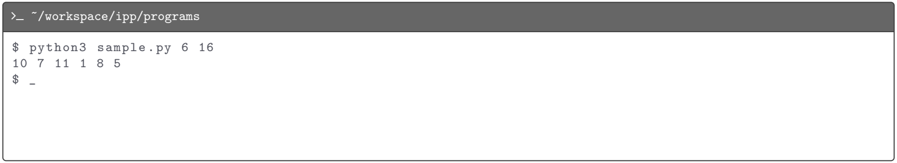
```
>_ ~/workspace/ipp/programs

$ python3 sample.py 6 16
```

Lists

Program: `sample.py`

- Command-line input: *m* (int) and *n* (int)
- Standard output: a random sample (without replacement) of *m* integers from the interval $[0, n)$



```
>_ ~/workspace/ipp/programs
```

```
$ python3 sample.py 6 16
10 7 11 1 8 5
$ _
```

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/ipp/programs

$ python3 sample.py 6 16
10 7 11 1 8 5
$ python3 sample.py 10 1000
```

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 sample.py 6 16
10 7 11 1 8 5
$ python3 sample.py 10 1000
258 802 440 28 244 256 564 11 515 24
$ _
```

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 sample.py 6 16
10 7 11 1 8 5
$ python3 sample.py 10 1000
258 802 440 28 244 256 564 11 515 24
$ python3 sample.py 20 20
```

Lists

Program: `sample.py`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/ipp/programs

$ python3 sample.py 6 16
10 7 11 1 8 5
$ python3 sample.py 10 1000
258 802 440 28 244 256 564 11 515 24
$ python3 sample.py 20 20
15 11 13 1 5 8 16 7 0 4 10 18 19 14 3 12 2 6 9 17
$ _
```

Lists

Lists

sample.py

```python
import stdarray
import stdio
import stdrandom
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 = stdrandom.uniformInt(i, n)
    temp = perm[r]
    perm[r] = perm[i]
    perm[i] = temp
for i in range(m):
    stdio.write(str(perm[i]) + ' ')
stdio.writeln()
```

Lists

Lists

Program: `couponcollector.py`

Lists

Program: `couponcollector.py`

- Command-line input: n (int)

Lists

Program: `couponcollector.py`

- Command-line input: *n* (int)
- Standard output: number of coupons one must collect before obtaining one of each of *n* types

Lists

Program: `couponcollector.py`

- Command-line input: *n* (int)
- Standard output: number of coupons one must collect before obtaining one of each of *n* types

```
>_ ~/workspace/ipp/programs
```

```
$ _
```

Lists

Program: `couponcollector.py`

- Command-line input: *n* (int)
- Standard output: number of coupons one must collect before obtaining one of each of *n* types

≻_ ~/workspace/ipp/programs

```
$ python3 couponcollector.py 1000
```

Lists

Program: `couponcollector.py`

- Command-line input: *n* (int)
- Standard output: number of coupons one must collect before obtaining one of each of *n* types

```
>_ ~/workspace/ipp/programs
```

```
$ python3 couponcollector.py 1000
6276
$ _
```

Lists

Program: `couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining one of each of n types

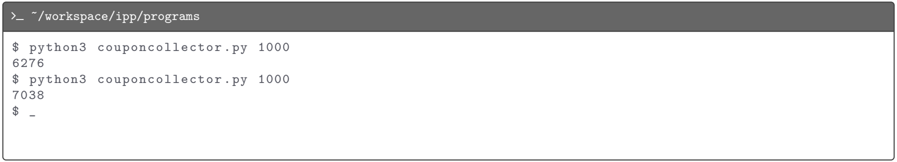
```
>_ ~/workspace/ipp/programs

$ python3 couponcollector.py 1000
6276
$ python3 couponcollector.py 1000
```

Lists

Program: `couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining one of each of n types



```
>_ ~/workspace/ipp/programs
```

```
$ python3 couponcollector.py 1000
6276
$ python3 couponcollector.py 1000
7038
$ _
```

Lists

Program: `couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining one of each of n types

```
>_ ~/workspace/ipp/programs
```

```
$ python3 couponcollector.py 1000
6276
$ python3 couponcollector.py 1000
7038
$ python3 couponcollector.py 1000000
```

Lists

Program: `couponcollector.py`

- Command-line input: *n* (int)
- Standard output: number of coupons one must collect before obtaining one of each of *n* types

```
>_ ~/workspace/ipp/programs

$ python3 couponcollector.py 1000
6276
$ python3 couponcollector.py 1000
7038
$ python3 couponcollector.py 1000000
13401736
$ _
```

Lists

Lists

couponcollector.py

```python
import stdarray
import stdio
import stdrandom
import sys

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

Lists

Lists

Program: `primesieve.py`

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)

Lists

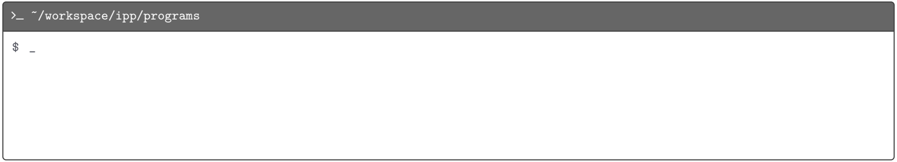
Program: `primesieve.py`

- Command-line input: n (int)
- Standard output: number of primes that are less than or equal to n

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)
- Standard output: number of primes that are less than or equal to *n*



```
>_ ~/workspace/ipp/programs
```

```
$ _
```

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)
- Standard output: number of primes that are less than or equal to *n*

```
>_ ~/workspace/ipp/programs

$ python3 primesieve.py 100
```

Lists

Program: `primesieve.py`

- Command-line input: n (int)
- Standard output: number of primes that are less than or equal to n

```
>_ ~/workspace/ipp/programs

$ python3 primesieve.py 100
25
$ _
```

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)
- Standard output: number of primes that are less than or equal to *n*

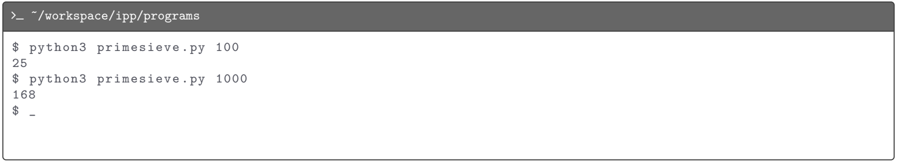
```
>_ ~/workspace/ipp/programs
```

```
$ python3 primesieve.py 100
25
$ python3 primesieve.py 1000
```

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)
- Standard output: number of primes that are less than or equal to *n*



```
>_ ~/workspace/ipp/programs
```

```
$ python3 primesieve.py 100
25
$ python3 primesieve.py 1000
168
$ _
```

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)
- Standard output: number of primes that are less than or equal to *n*

```
>_ ~/workspace/ipp/programs
```

```
$ python3 primesieve.py 100
25
$ python3 primesieve.py 1000
168
$ python3 primesieve.py 1000000
```

Lists

Program: `primesieve.py`

- Command-line input: *n* (int)
- Standard output: number of primes that are less than or equal to *n*

```
>_ ~/workspace/ipp/programs

$ python3 primesieve.py 100
25
$ python3 primesieve.py 1000
168
$ python3 primesieve.py 1000000
78498
$ _
```

Lists

Lists

primesieve.py

```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)
```

Lists

Lists

Creating a 2D list

```
<name> = [[<expression>, <expression>, ..., <expression>],
          [<expression>, <expression>, ..., <expression>],
          ...
          [<expression>, <expression>, ..., <expression>]]
```

Lists

Creating a 2D list

```
<name> = [[<expression>, <expression>, ..., <expression>],
          [<expression>, <expression>, ..., <expression>],
          ...
          [<expression>, <expression>, ..., <expression>]]
```

Example

```
a = [[ 1,  2,  3,  4],
     [ 5,  6,  7,  8],
     [ 9, 10, 11, 12]]
i = [[1, 0, 0],
     [0, 1, 0],
     [0, 0, 1]]
```

Lists

Lists

Appending to a 2D list

```
<name> += [<expression>]
```

Lists

Appending to a 2D list

```
<name> += [<expression>]
```

Example (creating a 2D list `a` with `m x n` zeros)

```
a = []
for i in range(m):
    row = stdarray.create1D(n, 0.0)
    a += [row]
```

Lists

Appending to a 2D list

```
<name> += [<expression>]
```

Example (creating a 2D list `a` with `m x n` zeros)

```
1  a = []
2  for i in range(m):
3      row = stdarray.create1D(n, 0.0)
4      a += [row]
```

Variable trace (`m = 2, n = 3`)

line #	a	i	row
1	[]		
2	[]	0	
3	[]	0	[0.0, 0.0, 0.0]
4	[[0.0, 0.0, 0.0]]	0	[0.0, 0.0, 0.0]
2	[[0.0, 0.0, 0.0]]	1	[0.0, 0.0, 0.0]
3	[[0.0, 0.0, 0.0]]	1	[0.0, 0.0, 0.0]
4	[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]	1	[0.0, 0.0, 0.0]

Lists

Lists

The object at row `i` and column `j` in a 2D list `<name>` with `m` rows and `n` columns is referred to as `<name>[i][j]` where `0 <= i < m` and `0 <= j < n`

Lists

The object at row `i` and column `j` in a 2D list `<name>` with `m` rows and `n` columns is referred to as `<name>[i][j]` where `0 <= i < m` and `0 <= j < n`

Example (adding two `n x 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]
```

Lists

The object at row `i` and column `j` in a 2D list `<name>` with `m` rows and `n` columns is referred to as `<name>[i][j]` where `0 <= i < m` and `0 <= j < n`

Example (adding two `n x n` matrices `a` and `b`)

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

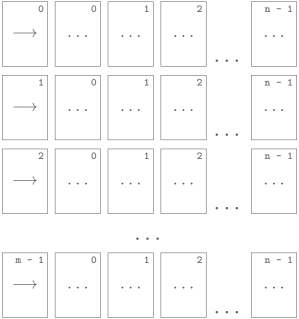
Variable trace (`a = [[1.0, 2.0], [3.0, 4.0]], b = [[2.0, 3.0], [4.0, 5.0]], n = 2`)

line #	c	i	j
1	[[0.0, 0.0], [0.0, 0.0]]		
2	[[0.0, 0.0], [0.0, 0.0]]	0	
3	[[0.0, 0.0], [0.0, 0.0]]	0	0
4	[[3.0, 0.0], [0.0, 0.0]]	0	0
3	[[0.0, 0.0], [0.0, 0.0]]	0	1
4	[[3.0, 5.0], [0.0, 0.0]]	0	1
2	[[3.0, 5.0], [0.0, 0.0]]	1	
3	[[3.0, 5.0], [0.0, 0.0]]	1	0
4	[[3.0, 5.0], [7.0, 0.0]]	1	0
3	[[3.0, 5.0], [7.0, 0.0]]	1	1
4	[[3.0, 5.0], [7.0, 9.0]]	1	1

Lists

Lists

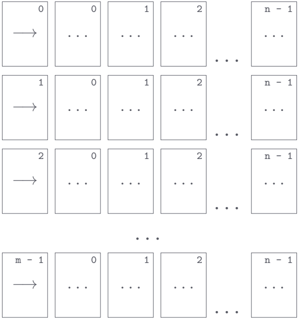
Memory model for a 2D list `<name>` with `m` rows and `n` columns



Note: `m` can be obtained as `len(<name>)` and `n` as `len(<name>[0])`

Lists

Memory model for a 2D list `<name>` with `m` rows and `n` columns

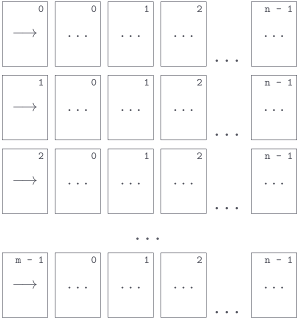


Note: `m` can be obtained as `len(<name>)` and `n` as `len(<name>[0])`

Index to row-major order: `k = n * i + j`

Lists

Memory model for a 2D list `<name>` with `m` rows and `n` columns



Note: `m` can be obtained as `len(<name>)` and `n` as `len(<name>[0])`

Index to row-major order: `k = n * i + j`

Row-major order to index: `i = k // n` and `j = k % n`

Lists

Lists

Program: `selfavoid.py`

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)

Lists

Program: `selfavoid.py`

- Command-line input: n (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/ipp/programs

$ _
```

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/ipp/programs

$ python3 selfavoid.py 20 1000
```

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/ipp/programs

$ python3 selfavoid.py 20 1000
33% dead ends
$ _
```

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

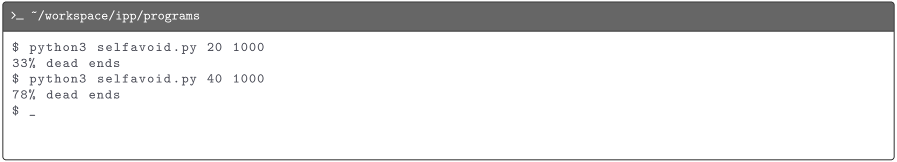
```
>_ ~/workspace/ipp/programs

$ python3 selfavoid.py 20 1000
33% dead ends
$ python3 selfavoid.py 40 1000
```

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice



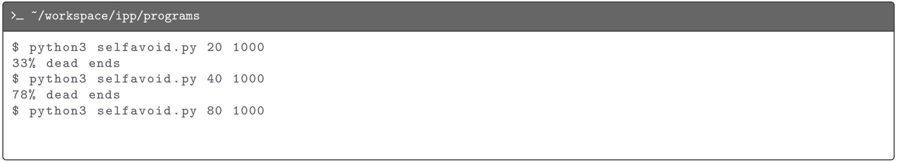
```
>_ ~/workspace/ipp/programs
```

```
$ python3 selfavoid.py 20 1000
33% dead ends
$ python3 selfavoid.py 40 1000
78% dead ends
$ _
```

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice



```
>_ ~/workspace/ipp/programs
```

```
$ python3 selfavoid.py 20 1000
33% dead ends
$ python3 selfavoid.py 40 1000
78% dead ends
$ python3 selfavoid.py 80 1000
```

Lists

Program: `selfavoid.py`

- Command-line input: *n* (int) and *trials* (int)
- Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

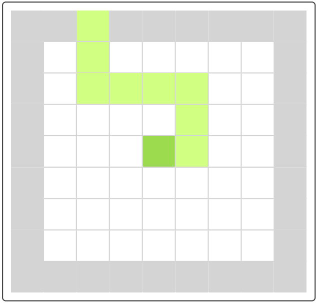
~/workspace/ipp/programs

```
$ python3 selfavoid.py 20 1000
33% dead ends
$ python3 selfavoid.py 40 1000
78% dead ends
$ python3 selfavoid.py 80 1000
98% dead ends
$ _
```

Lists

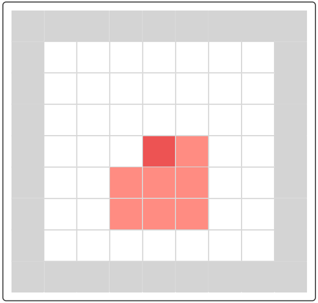
Lists

Escape



→ ↑ ↑ ← ← ← ↑ ↑

Dead End



→ ↓ ↓ ← ← ↑ →

Lists

Lists

selfavoid.py

```
import stdarray
import stdio
import stdrandom
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 = stdrandom.uniformInt(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

Lists

A 2D list with rows of nonuniform length is called a ragged list

Lists

A 2D list with rows of nonuniform length is called a ragged list

Example (writing a 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()
```

Lists

A 2D list with rows of nonuniform length is called a ragged list

Example (writing a 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()
```

Output when `a = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]`

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

Tuples

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> _
```

Tuples

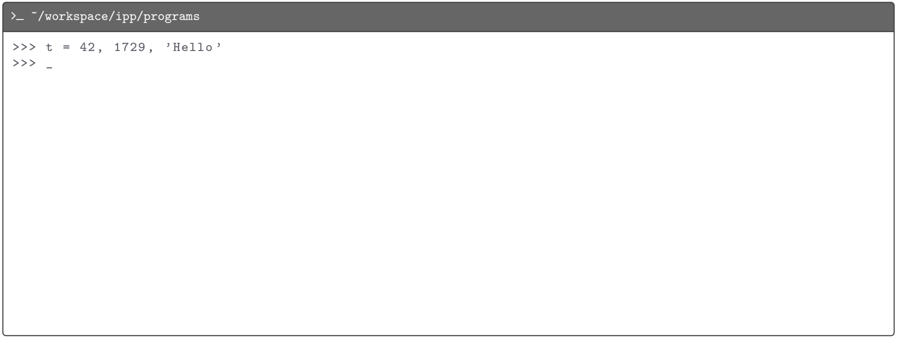
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, 'Hello'
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



```
>>> t = 42, 1729, 'Hello'
>>> _
```

Tuples

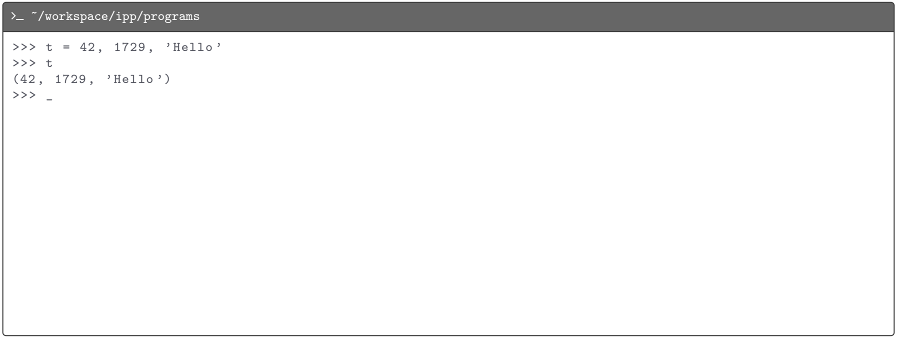
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs

>>> t = 42, 1729, 'Hello'
>>> t
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, 'Hello'
>>> t
(42, 1729, 'Hello')
>>> _
```

Tuples

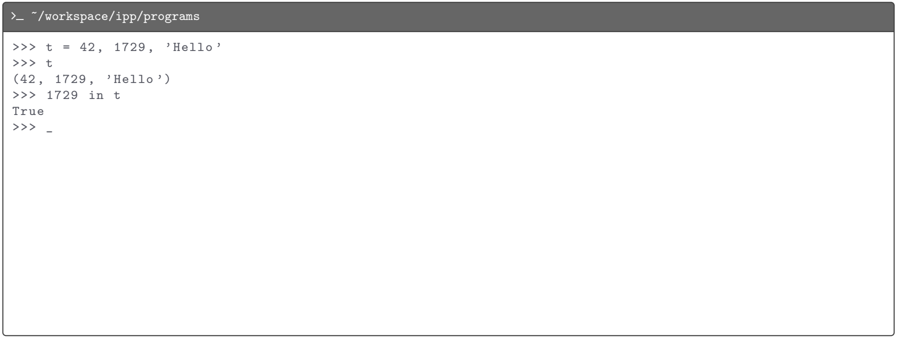
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, 'Hello'
>>> t
(42, 1729, 'Hello')
>>> 1729 in t
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, 'Hello'
>>> t
(42, 1729, 'Hello')
>>> 1729 in t
True
>>> _
```

Tuples

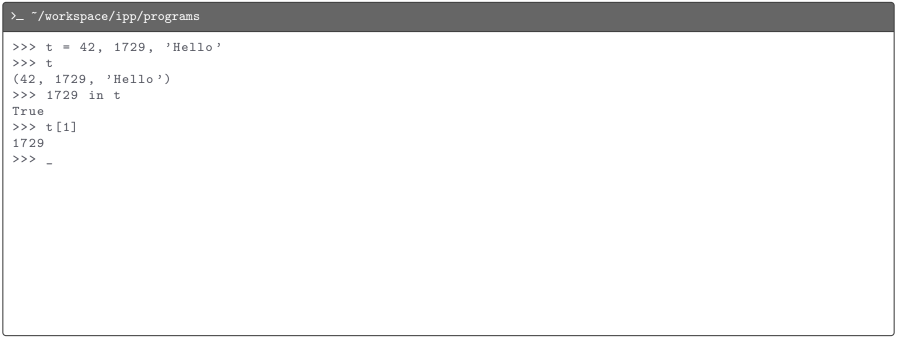
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, 'Hello'
>>> t
(42, 1729, 'Hello')
>>> 1729 in t
True
>>> t[1]
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, 'Hello'
>>> t
(42, 1729, 'Hello')
>>> 1729 in t
True
>>> t[1]
1729
>>> _
```

Tuples

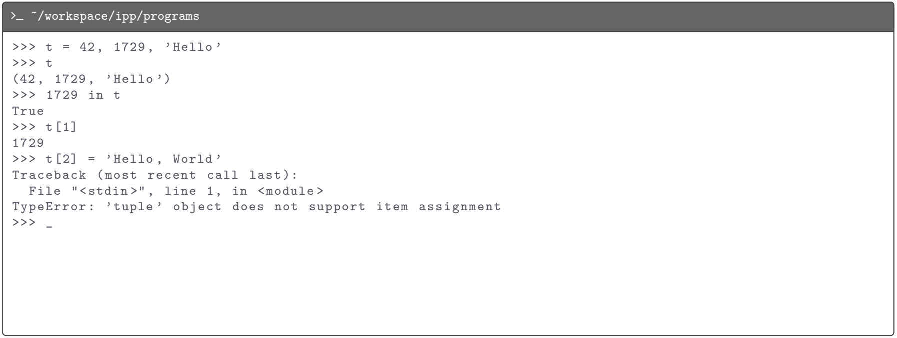
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> _
```

Tuples

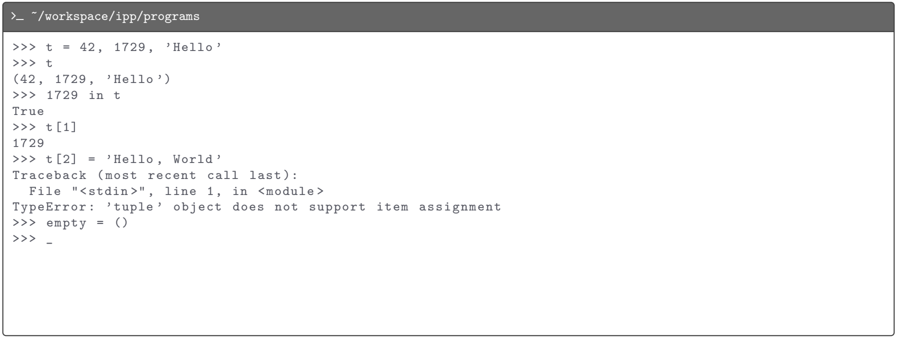
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



>_ ~/workspace/ipp/programs

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> _
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> len(empty)
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> len(empty)
0
>>> _
```

Tuples

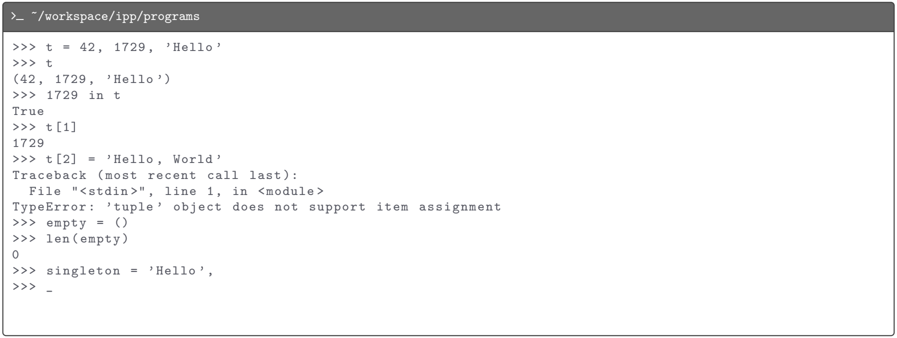
A tuple (object of type `tuple`) is an immutable, ordered collection of objects

›_ ~/workspace/ipp/programs

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> len(empty)
0
>>> singleton = ’Hello’,
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects



>_ ~/workspace/ipp/programs

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> len(empty)
0
>>> singleton = ’Hello’,
>>> _
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> len(empty)
0
>>> singleton = ’Hello’,
>>> len(singleton)
```

Tuples

A tuple (object of type `tuple`) is an immutable, ordered collection of objects

```
>_ ~/workspace/ipp/programs
```

```
>>> t = 42, 1729, ’Hello’
>>> t
(42, 1729, ’Hello’)
>>> 1729 in t
True
>>> t[1]
1729
>>> t[2] = ’Hello, World’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
>>> empty = ()
>>> len(empty)
0
>>> singleton = ’Hello’,
>>> len(singleton)
1
>>> _
```

Sets

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> _
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = ['orange', 'apple', 'pear', 'orange', 'banana', 'apple']
```

Sets

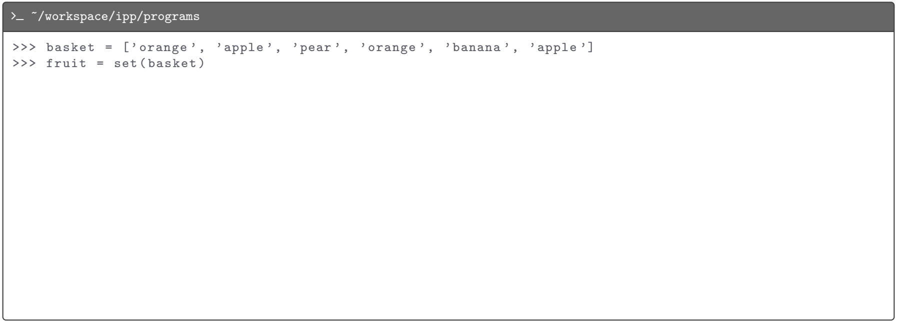
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs

>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> _
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



```
>_ ~/workspace/ipp/programs
```

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

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs

>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> _
```

Sets

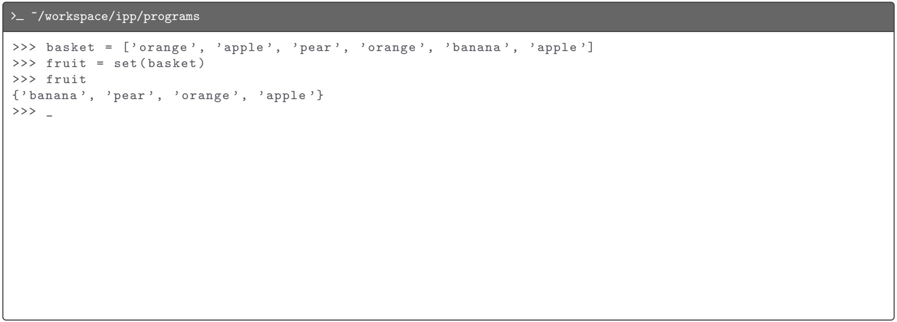
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

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

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



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

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

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

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs

>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> _
```

Sets

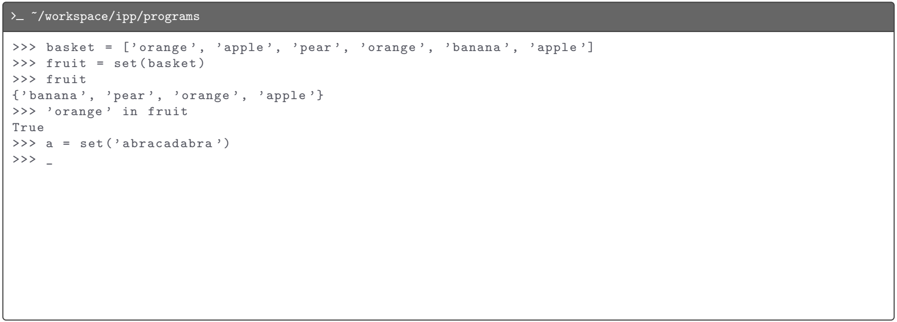
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> _
```

Sets

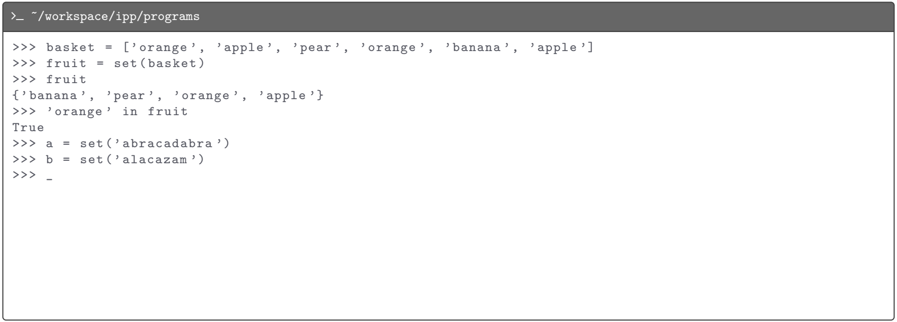
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> _
```

Sets

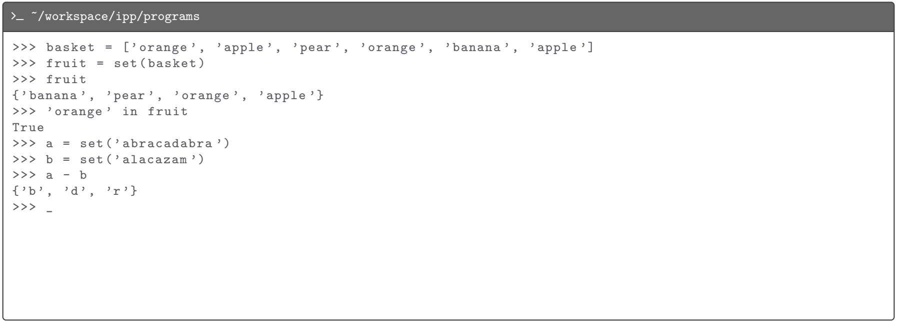
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



```
>>> basket = ['orange', 'apple', 'pear', 'orange', 'banana', 'apple']
>>> fruit = set(basket)
>>> fruit
{'banana', 'pear', 'orange', 'apple'}
>>> 'orange' in fruit
True
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a - b
{'b', 'd', 'r'}
>>> _
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
{’b’, ’d’, ’r’}
>>> a | b
```

Sets

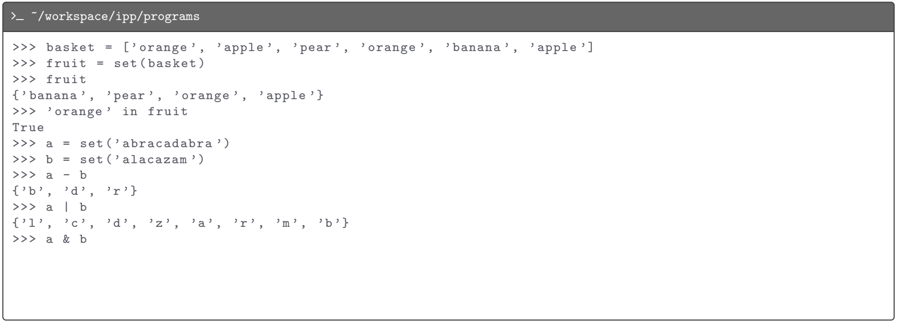
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
{’b’, ’d’, ’r’}
>>> a | b
{’l’, ’c’, ’d’, ’z’, ’a’, ’r’, ’m’, ’b’}
>>> _
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
{’b’, ’d’, ’r’}
>>> a | b
{’l’, ’c’, ’d’, ’z’, ’a’, ’r’, ’m’, ’b’}
>>> a & b
```

Sets

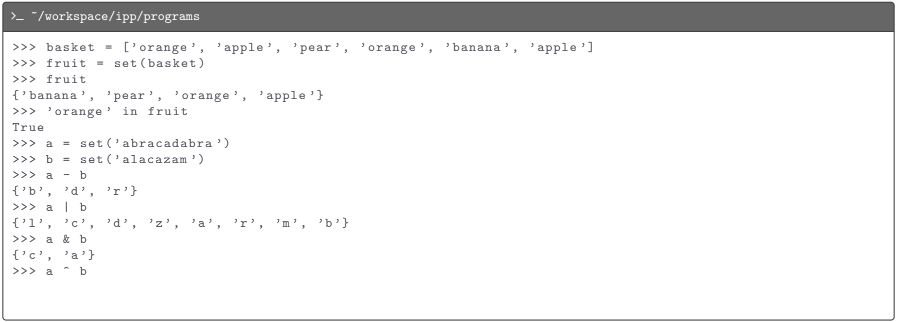
A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
{’b’, ’d’, ’r’}
>>> a | b
{’l’, ’c’, ’d’, ’z’, ’a’, ’r’, ’m’, ’b’}
>>> a & b
{’c’, ’a’}
>>> _
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates



```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
{’b’, ’d’, ’r’}
>>> a | b
{’l’, ’c’, ’d’, ’z’, ’a’, ’r’, ’m’, ’b’}
>>> a & b
{’c’, ’a’}
>>> a ^ b
```

Sets

A set (object of type `set`) is an unordered collection of objects with no duplicates

```
>_ ~/workspace/ipp/programs
```

```
>>> basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
>>> fruit = set(basket)
>>> fruit
{’banana’, ’pear’, ’orange’, ’apple’}
>>> ’orange’ in fruit
True
>>> a = set(’abracadabra’)
>>> b = set(’alacazam’)
>>> a - b
{’b’, ’d’, ’r’}
>>> a | b
{’l’, ’c’, ’d’, ’z’, ’a’, ’r’, ’m’, ’b’}
>>> a & b
{’c’, ’a’}
>>> a ^ b
{’l’, ’r’, ’d’, ’m’, ’b’, ’z’}
>>> _
```

Dictionaries

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

>>> _
```

Dictionaries

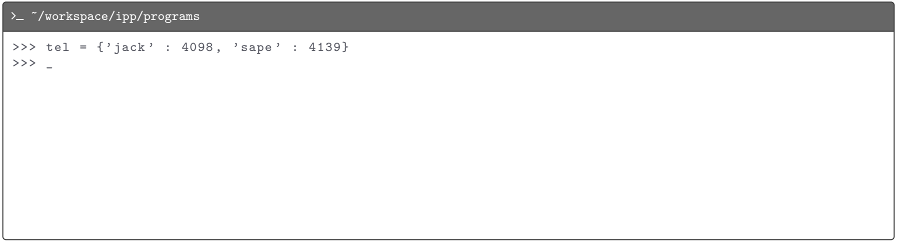
A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

>>> tel = {'jack' : 4098, 'sape' : 4139}
```

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique



```
>_ ~/workspace/ipp/programs
```

```
>>> tel = {’jack’ : 4098, ’sape’ : 4139}
>>> _
```

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs
```

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

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

>>> tel = {'jack' : 4098, 'sape' : 4139}
>>> tel['guido'] = 4127
>>> _
```

Dictionaries

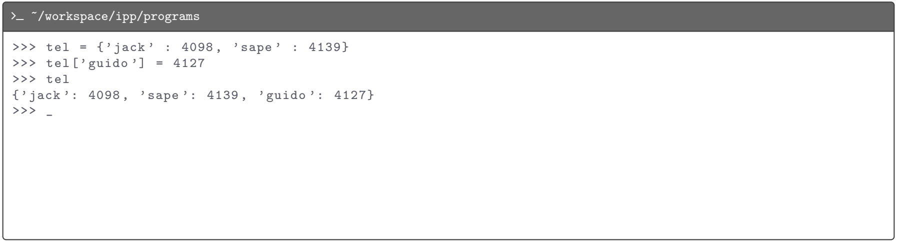
A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

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

Dictionaries

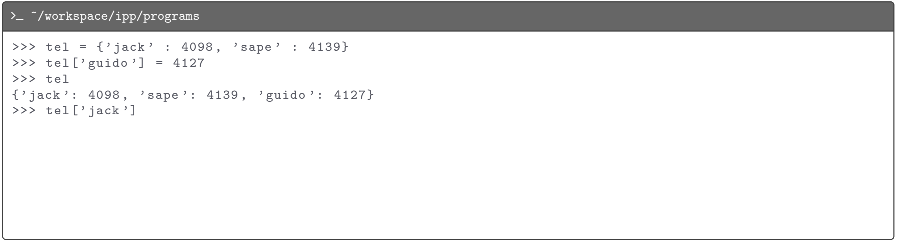
A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique



```
>>> tel = {’jack’ : 4098, ’sape’ : 4139}
>>> tel[’guido’] = 4127
>>> tel
{’jack’: 4098, ’sape’: 4139, ’guido’: 4127}
>>> _
```

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique



>_ ~/workspace/ipp/programs

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

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs
```

```
>>> tel = {’jack’ : 4098, ’sape’ : 4139}
>>> tel[’guido’] = 4127
>>> tel
{’jack’: 4098, ’sape’: 4139, ’guido’: 4127}
>>> tel[’jack’]
4098
>>> _
```

Dictionaries

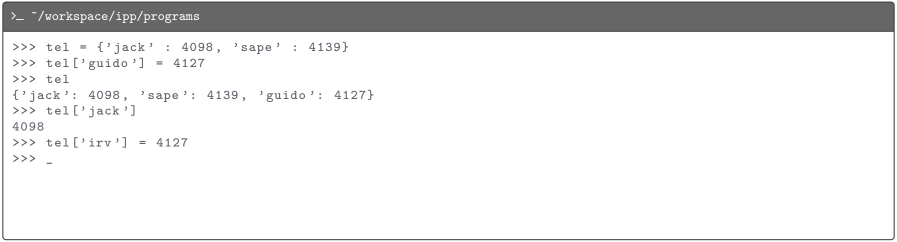
A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

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

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique



```
>>> tel = {’jack’ : 4098, ’sape’ : 4139}
>>> tel[’guido’] = 4127
>>> tel
{’jack’: 4098, ’sape’: 4139, ’guido’: 4127}
>>> tel[’jack’]
4098
>>> tel[’irv’] = 4127
>>> _
```

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

>>> tel = {’jack’ : 4098, ’sape’ : 4139}
>>> tel[’guido’] = 4127
>>> tel
{’jack’: 4098, ’sape’: 4139, ’guido’: 4127}
>>> tel[’jack’]
4098
>>> tel[’irv’] = 4127
>>> tel
```

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

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

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

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

Dictionaries

A dictionary (object of type `dict`) is an unordered collection of key-value pairs (each an object), with the keys being unique

```
>_ ~/workspace/ipp/programs

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

Advanced Looping Techniques

Advanced Looping Techniques

You can loop over a sequence with access to both index and value using `enumerate()`

Advanced Looping Techniques

You can loop over a sequence with access to both index and value using `enumerate()`

Example

```
for i, v in enumerate([’tic’, ’tac’, ’toe’]):
    stdio.writeln(str(i) + ’ ’ + v)
```

Advanced Looping Techniques

You can loop over a sequence with access to both index and value using `enumerate()`

Example

```
for i, v in enumerate([’tic’, ’tac’, ’toe’]):
    stdio.writeln(str(i) + ’ ’ + v)
```

```
0 tic
1 tac
2 toe
```

Advanced Looping Techniques

Advanced Looping Techniques

You can loop over two or more equal-length sequences at the same time using `zip()`

Advanced Looping Techniques

You can loop over two or more equal-length sequences at the same time using `zip()`

Example

```
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 + ’.’)
```

Advanced Looping Techniques

You can loop over two or more equal-length sequences at the same time using `zip()`

Example

```
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.
```

Advanced Looping Techniques

Advanced Looping Techniques

You can loop over a sequence in reverse using `reversed()`

Advanced Looping Techniques

You can loop over a sequence in reverse using `reversed()`

Example

```python
for i in reversed(range(1, 10, 2)):
    stdio.writeln(i)
```

Advanced Looping Techniques

You can loop over a sequence in reverse using `reversed()`

Example

```
for i in reversed(range(1, 10, 2)):
    stdio.writeln(i)
```

```
9
7
5
3
1
```

Advanced Looping Techniques

Advanced Looping Techniques

You can loop over a sequence in sorted order using `sorted()`

Advanced Looping Techniques

You can loop over a sequence in sorted order using `sorted()`

Example

```
basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
for fruit in sorted(basket):
   stdio.writeln(fruit)
```

Advanced Looping Techniques

You can loop over a sequence in sorted order using `sorted()`

Example

```
basket = [’orange’, ’apple’, ’pear’, ’orange’, ’banana’, ’apple’]
for fruit in sorted(basket):
   stdio.writeln(fruit)
```

```
apple
apple
banana
orange
orange
pear
```