Introduction to Programming in Python

Imperative Programming: Collection Data Types

Outline

One Dimensional (1D) Lists

One Dimensional (1D) Lists

A list (object of type `list`), also known as an array, is an ordered collection of objects

One Dimensional (1D) Lists

A list (object of type `list`), also known as an array, is an ordered collection of objects

Creating a 1D list

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

One Dimensional (1D) Lists

A list (object of type `list`), also known as an array, is an ordered collection of objects

Creating a 1D list

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

Example

```
suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
powersOfTwo = [0.0625, 0.125, 0.25, 0.5, 1.0, 2.0, 4.0, 8.0, 16.0, 32.0]
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Appending to a 1D list

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

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
1	[]	

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
2	[]	0

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
3	[0]	0

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
2	[0]	1

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
3	[0, 0]	1

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
2	[0, 0]	2

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

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

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

```
a = []
for i in range(3):
    a += [0]
```

line #	a	i
2	[0, 0, 0]	

One Dimensional (1D) Lists

Appending to a 1D list

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

Example (creating a 1D list a with 3 zeros)

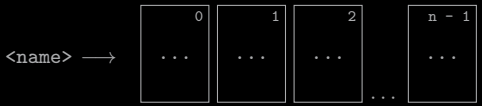
```
a = []
for i in range(3):
    a += [0]
```

line #	a	i

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Memory model



One Dimensional (1D) Lists

One Dimensional (1D) Lists

The `stdarray` library provides a function for creating 1D lists

`create1D(n, value = None)`	creates and returns a 1D list of size `n`, with each element initialized to `value`

One Dimensional (1D) Lists

The `stdarray` library provides a function for creating 1D lists

`create1D(n, value = None)`	creates and returns a 1D list of size `n`, with each element initialized to `value`

Python has several built-in functions that operate on lists

One Dimensional (1D) Lists

The `stdarray` library provides a function for creating 1D lists

`create1D(n, value = None)`	creates and returns a 1D list of size `n`, with each element initialized to `value`

Python has several built-in functions that operate on lists

For example, given a list `x`

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

One Dimensional (1D) Lists

One Dimensional (1D) Lists

The ith element in a list `x` is accessed as `x[i]`, where `0 <= i < len(x)`

One Dimensional (1D) Lists

The ith element in a list x is accessed as x[i], where 0 <= i < len(x)

The ith element in a list x is assigned a value as

```
x[i] = <expression>
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
a = [2.0, 4.0, 6.0]

total = 0.0
for i in range(len(a)):
    total += a[i]

avg = total / len(a)
```

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
1			

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
3	0.0		

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
4	0.0	0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
5	2.0	0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
4	2.0	1	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
5	6.0	1	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
4	6.0	2	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
5	12.0	2	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
4	12.0		

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg
7	12.0		4.0

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for i in range(len(a)):
5      total += a[i]
6
7  avg = total / len(a)
```

line #	total	i	avg

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
a = [2.0, 4.0, 6.0]

total = 0.0
for v in a:
    total += v

avg = total / len(a)
```

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
1			

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
3	0.0		

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
4	0.0	2.0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
5	2.0	2.0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
4	2.0	4.0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
5	6.0	4.0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
4	6.0	6.0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
5	12.0	6.0	

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
4	12.0		

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg
7	12.0		4.0

One Dimensional (1D) Lists

Example (Averaging Numbers in a List)

```
1  a = [2.0, 4.0, 6.0]
2
3  total = 0.0
4  for v in a:
5      total += v
6
7  avg = total / len(a)
```

line #	total	v	avg

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
1		

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
3	[0, 0, 0, 0, 0]	

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
5	[0, 0, 0, 0, 0]	0

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
6	[0, 0, 0, 0, 0]	0

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
5	[0, 0, 0, 0, 0]	1

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
6	[0, 1, 0, 0, 0]	1

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
5	[0, 1, 0, 0, 0]	2

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
6	[0, 1, 2, 0, 0]	2

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
5	[0, 1, 2, 0, 0]	3

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
6	[0, 1, 2, 3, 0]	3

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

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

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i
6	[0, 1, 2, 3, 4]	4

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

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

One Dimensional (1D) Lists

Example (Identity Permutation)

```
1  import stdarray
2
3  perm = stdarray.create1D(5, 0)
4
5  for i in range(5):
6      perm[i] = i
```

line #	perm	i

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
1	[0,1,2,3,4]		

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1 perm = [0, 1, 2, 3, 4]
2
3 for i in range(5 // 2):
4     temp = perm[i]
5     perm[i] = perm[4 - i]
6     perm[4 - i] = temp
```

line #	perm	i	temp
3	[0,1,2,3,4]	0	

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
4	[0,1,2,3,4]	0	0

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
5	[4,1,2,3,4]	0	0

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1 perm = [0, 1, 2, 3, 4]
2
3 for i in range(5 // 2):
4     temp = perm[i]
5     perm[i] = perm[4 - i]
6     perm[4 - i] = temp
```

line #	perm	i	temp
6	[4,1,2,3,0]	0	0

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
3	[4,1,2,3,0]	1	

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
4	[4,1,2,3,0]	1	1

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
5	[4,3,2,3,0]	1	1

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
6	[4,3,2,1,0]	1	1

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp
3	[4,3,2,1,0]	2	

One Dimensional (1D) Lists

Example (In-place Reversal)

```
1  perm = [0, 1, 2, 3, 4]
2
3  for i in range(5 // 2):
4      temp = perm[i]
5      perm[i] = perm[4 - i]
6      perm[4 - i] = temp
```

line #	perm	i	temp

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
1			

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
3	[0,1,2,3,4]		

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
5	[0,1,2,3,4]	0	

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
6	[0,1,2,3,4]	0	3

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
7	[0,1,2,3,4]	0	3

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
8	[0,1,2,0,4]	0	3

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
9	[3,1,2,0,4]	0	3

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
5	[3,1,2,0,4]	1	

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
6	[3,1,2,0,4]	1	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
7	[3,1,2,0,4]	1	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
8	[3,1,1,0,4]	1	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
9	[3,2,1,0,4]	1	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
5	[3,2,1,0,4]	2	

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
6	[3,2,1,0,4]	2	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
7	[3,2,1,0,4]	2	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
8	[3,2,1,0,4]	2	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
9	[3,2,1,0,4]	2	2

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
5	[3,2,1,0,4]	3	

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
6	[3,2,1,0,4]	3	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
7	[3,2,1,0,4]	3	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
8	[3,2,1,0,0]	3	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
9	[3,2,1,4,0]	3	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
5	[3,2,1,4,0]	4	

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
6	[3,2,1,4,0]	4	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
7	[3,2,1,4,0]	4	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
8	[3,2,1,4,0]	4	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
9	[3,2,1,4,0]	4	4

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r
5	[3,2,1,4,0]	5	

One Dimensional (1D) Lists

Example (Knuth Shuffle)

```
1  import StdRandom
2
3  perm = [0, 1, 2, 3, 4]
4
5  for i in range(5):
6      r = StdRandom.uniform(i, 5)
7      temp = perm[r]
8      perm[r] = perm[i]
9      perm[i] = temp
```

line #	perm	i	r

One Dimensional (1D) Lists

One Dimensional (1D) Lists

`sample.py`

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

One Dimensional (1D) Lists

`sample.py`

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

```
× ~/workspace/ipp
$ _
```

One Dimensional (1D) Lists

`sample.py`

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

```
× ~/workspace/ipp
$ python3 sample.py 6 16
```

One Dimensional (1D) Lists

`sample.py`

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

× ~/workspace/ipp

```
$ python3 sample.py 6 16
11 10 12 13 6 8
$ _
```

One Dimensional (1D) Lists

`sample.py`

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

× ~/workspace/ipp

```
$ python3 sample.py 6 16
11 10 12 13 6 8
$ python3 sample.py 10 1000
```

One Dimensional (1D) Lists

`sample.py`

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

× ~/workspace/ipp

```
$ python3 sample.py 6 16
11 10 12 13 6 8
$ python3 sample.py 10 1000
21 432 270 287 166 484 437 675 78 213
$ _
```

One Dimensional (1D) Lists

`sample.py`

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

```
× ~/workspace/ipp
$ python3 sample.py 6 16
11 10 12 13 6 8
$ python3 sample.py 10 1000
21 432 270 287 166 484 437 675 78 213
$ python3 sample.py 20 20


```

One Dimensional (1D) Lists

`sample.py`

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

× ~/workspace/ipp

```
$ python3 sample.py 6 16
11 10 12 13 6 8
$ python3 sample.py 10 1000
21 432 270 287 166 484 437 675 78 213
$ python3 sample.py 20 20
9 0 15 13 4 8 11 17 3 18 16 5 7 19 14 12 2 1 10 6
$ _
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9
perm[i]	0	0	0	0	0	0	0	0	0	0

m = 5, n = 10

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9
perm[i]	0	1	2	3	4	5	6	7	8	9

m = 5, n = 10

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9
perm[i]	3	7	2	9	1	5	6	4	8	0

m = 5, n = 10

One Dimensional (1D) Lists

One Dimensional (1D) Lists

× sample.py

```
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()
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

`couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

One Dimensional (1D) Lists

`couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

```
× ~/workspace/ipp
$ _
```

One Dimensional (1D) Lists

`couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

```
× ~/workspace/ipp
$ python3 couponcollector.py 1000
```

One Dimensional (1D) Lists

couponcollector.py

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

× ~/workspace/ipp

```
$ python3 couponcollector.py 1000
8317
$ _
```

One Dimensional (1D) Lists

couponcollector.py

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

× ~/workspace/ipp

```
$ python3 couponcollector.py 1000
8317
$ python3 couponcollector.py 1000
```

One Dimensional (1D) Lists

`couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

× ~/workspace/ipp

```
$ python3 couponcollector.py 1000
8317
$ python3 couponcollector.py 1000
7867
$ _
```

One Dimensional (1D) Lists

`couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

× ~/workspace/ipp

```
$ python3 couponcollector.py 1000
8317
$ python3 couponcollector.py 1000
7867
$ python3 couponcollector.py 1000000


```

One Dimensional (1D) Lists

`couponcollector.py`

- Command-line input: n (int)
- Standard output: number of coupons one must collect before obtaining at least one of the n unique coupons

× ~/workspace/ipp

```
$ python3 couponcollector.py 1000
8317
$ python3 couponcollector.py 1000
7867
$ python3 couponcollector.py 1000000
15942756
$ _
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

value	count	collectedCount
	0	0

value	0	1	2
isCollected[value]	F	F	F

One Dimensional (1D) Lists

value	count	collectedCount
1	1	1

value	0	1	2
isCollected[value]	F	T	F

One Dimensional (1D) Lists

value	count	collectedCount
1	2	1

value	0	1	2
isCollected[value]	F	T	F

One Dimensional (1D) Lists

value	count	collectedCount
1	3	1

value	0	1	2
isCollected[value]	F	T	F

One Dimensional (1D) Lists

value	count	collectedCount
2	4	2

value	0	1	2
isCollected[value]	F	T	T

One Dimensional (1D) Lists

value	count	collectedCount
0	5	3

value	0	1	2
isCollected[value]	T	T	T

One Dimensional (1D) Lists

One Dimensional (1D) Lists

× couponcollector.py

```
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)
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

`primesieve.py`

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

One Dimensional (1D) Lists

`primesieve.py`

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

```
× ~/workspace/ipp
$ _
```

One Dimensional (1D) Lists

`primesieve.py`

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

```
× ~/workspace/ipp
$ python3 primesieve.py 10
```

One Dimensional (1D) Lists

`primesieve.py`

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

× ~/workspace/ipp

```
$ python3 primesieve.py 10
4
$ _
```

One Dimensional (1D) Lists

`primesieve.py`

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

× ~/workspace/ipp

```
$ python3 primesieve.py 10
4
$ python3 primesieve.py 100
```

One Dimensional (1D) Lists

`primesieve.py`

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

× ~/workspace/ipp

```
$ python3 primesieve.py 10
4
$ python3 primesieve.py 100
25
$ _
```

One Dimensional (1D) Lists

`primesieve.py`

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

× ~/workspace/ipp

```
$ python3 primesieve.py 10
4
$ python3 primesieve.py 100
25
$ python3 primesieve.py 1000


```

One Dimensional (1D) Lists

`primesieve.py`

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

× ~/workspace/ipp

```
$ python3 primesieve.py 10
4
$ python3 primesieve.py 100
25
$ python3 primesieve.py 1000
168
$ _
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	F	F	F	F	F	F	F	F	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	T	T	T	T	T	T	T

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	T	T	T	T	T	T	T

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	T	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	T	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	F	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	F	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	F	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	F	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	F	F

One Dimensional (1D) Lists

i	0	1	2	3	4	5	6	7	8	9	10
isPrime[i]	F	F	T	T	F	T	F	T	F	F	F

One Dimensional (1D) Lists

One Dimensional (1D) Lists

× primesieve.py

```
1  import stdarray
2  import stdio
3  import sys
4
5  n = int(sys.argv[1])
6  isPrime = stdarray.create1D(n + 1, False)
7  for i in range(2, n + 1):
8      isPrime[i] = True
9  for i in range(2, n):
10     if isPrime[i]:
11         for j in range(2, n // i + 1):
12             isPrime[i * j] = False
13 count = 0
14 for i in range(2, n + 1):
15     count += 1 if isPrime[i] else 0
16 stdio.writeln(count)
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

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

One Dimensional (1D) Lists

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

Example

```
import stdio

x = [1, 3, 7]
y = x
x[1] = 42

stdio.writeln(x)
stdio.writeln(y)
```

writes

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

One Dimensional (1D) Lists

One Dimensional (1D) Lists

Example (creating a copy of a list `x`)

- Method 1 (using a loop)

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

One Dimensional (1D) Lists

Example (creating a copy of a list `x`)

- Method 1 (using a loop)

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

- Method 2 (using the slicing operator)

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

One Dimensional (1D) Lists

Example (creating a copy of a list `x`)

- Method 1 (using a loop)

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

- Method 2 (using the slicing operator)

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

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

One Dimensional (1D) Lists

One Dimensional (1D) Lists

- Strings can be manipulated like lists

One Dimensional (1D) Lists

Strings can be manipulated like lists

Example

```
import stdio

s = "Hello, World!"
for c in s[7 : len(s) - 1]:
    stdio.write(c)
stdio.writeln()
```

writes

```
World
```

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

Creating a 2D list

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

Two Dimensional (2D) Lists

Creating a 2D list

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

Example

```
identity = [[1, 0, 0],
            [0, 1, 0],
            [0, 0, 1]]

pascal = [[1, 0, 0, 0, 0],
          [1, 1, 0, 0, 0],
          [1, 2, 1, 0, 0],
          [1, 3, 3, 1, 0],
          [1, 4, 6, 4, 1]]
```

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

Appending to a 2D list

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
1		

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
3	[]	

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
4	[]	0

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
5	[[0,0]]	0

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
4	[[0,0]]	1

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
5	[[0,0],[0,0]]	1

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
4	[[0,0],[0,0]]	2

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
5	[[0,0],[0,0],[0,0]]	2

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i
4	[[0,0],[0,0],[0,0]]	

Two Dimensional (2D) Lists

Appending to a 2D list

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

Example (creating a list 2 x 3 list a with 6 zeros)

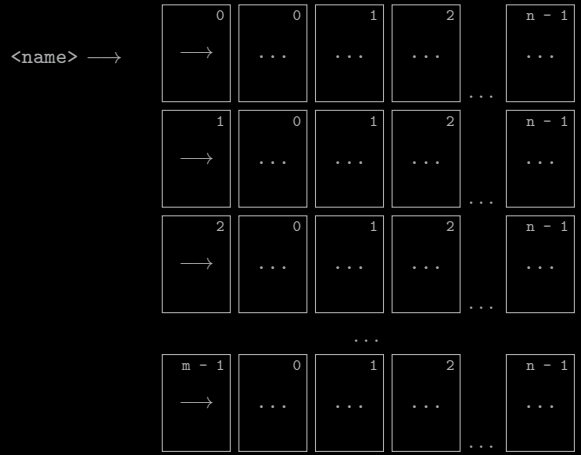
```
import stdarray

a = []
for i in range(3):
    a += [stdarray.create1D(2, 0)]
```

line #	a	i

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists



Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

The `stdarray` library also provides a function for creating 2D lists

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

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

The number of rows (say `m`) in a list `x` is obtained as `len(x)`

Two Dimensional (2D) Lists

- The number of rows (say `m`) in a list `x` is obtained as `len(x)`
- The number of columns (say `n`) in a list `x` is obtained as `len(x[0])`

Two Dimensional (2D) Lists

- The number of rows (say `m`) in a list `x` is obtained as `len(x)`
- The number of columns (say `n`) in a list `x` is obtained as `len(x[0])`
- The element in row `i` and column `j` of a list `x` is accessed as `x[i][j]`, where `0 <= i < m` and `0 <= j < n`

Two Dimensional (2D) Lists

The number of rows (say `m`) in a list `x` is obtained as `len(x)`

The number of columns (say `n`) in a list `x` is obtained as `len(x[0])`

The element in row `i` and column `j` of a list `x` is accessed as `x[i][j]`, where `0 <= i < m` and `0 <= j < n`

The element in row `i` and column `j` of a list `x` is assigned a value as

```
x[i][j] = <expression>
```

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
1			

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1 a = [[1, 2], [3, 4]]
2 b = [[2, 3], [4, 5]]
3
4 c = [[0, 0], [0, 0]]
5 for i in range(2):
6     for j in range(2):
7         c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
2			

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
4	[[0,0],[0,0]]		

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1 a = [[1, 2], [3, 4]]
2 b = [[2, 3], [4, 5]]
3
4 c = [[0, 0], [0, 0]]
5 for i in range(2):
6     for j in range(2):
7         c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
5	[[0,0],[0,0]]	0	

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
6	[[0,0],[0,0]]	0	0

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
7	[[3,0],[0,0]]	0	0

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1 a = [[1, 2], [3, 4]]
2 b = [[2, 3], [4, 5]]
3
4 c = [[0, 0], [0, 0]]
5 for i in range(2):
6     for j in range(2):
7         c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
6	[[3,0],[0,0]]	0	1

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
7	[[3,5],[0,0]]	0	1

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
6	[[3,5],[0,0]]	0	

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
5	[[3,5],[0,0]]	1	

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1 a = [[1, 2], [3, 4]]
2 b = [[2, 3], [4, 5]]
3
4 c = [[0, 0], [0, 0]]
5 for i in range(2):
6     for j in range(2):
7         c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
6	[[3,5],[0,0]]	1	0

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1 a = [[1, 2], [3, 4]]
2 b = [[2, 3], [4, 5]]
3
4 c = [[0, 0], [0, 0]]
5 for i in range(2):
6     for j in range(2):
7         c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
7	[[3,5],[7,0]]	1	0

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
6	[[3,5],[7,0]]	1	1

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
7	[[3,5],[7,9]]	1	1

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
6	[[3,5],[7,9]]	1	

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j
5	[[3,5],[7,9]]		

Two Dimensional (2D) Lists

Example (Matrix Addition)

```
1  a = [[1, 2], [3, 4]]
2  b = [[2, 3], [4, 5]]
3
4  c = [[0, 0], [0, 0]]
5  for i in range(2):
6      for j in range(2):
7          c[i][j] = a[i][j] + b[i][j]
```

line #	c	i	j

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

`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

Two Dimensional (2D) Lists

`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
$ _
```

Two Dimensional (2D) Lists

`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
$ python3 selfavoid.py 20 1000
```

Two Dimensional (2D) Lists

`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
$ python3 selfavoid.py 20 1000
32% dead ends
$ _
```

Two Dimensional (2D) Lists

`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
$ python3 selfavoid.py 20 1000
32% dead ends
$ python3 selfavoid.py 40 1000
```

Two Dimensional (2D) Lists

`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
$ python3 selfavoid.py 20 1000
32% dead ends
$ python3 selfavoid.py 40 1000
75% dead ends
$ _
```

Two Dimensional (2D) Lists

`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

```
$ python3 selfavoid.py 20 1000
32% dead ends
$ python3 selfavoid.py 40 1000
75% dead ends
$ python3 selfavoid.py 80 1000
```

Two Dimensional (2D) Lists

`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

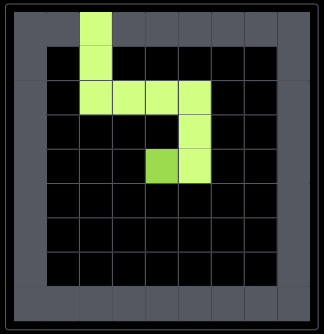
× ~/workspace/ipp

```
$ python3 selfavoid.py 20 1000
32% dead ends
$ python3 selfavoid.py 40 1000
75% dead ends
$ python3 selfavoid.py 80 1000
98% dead ends
$ _
```

Two Dimensional (2D) Lists

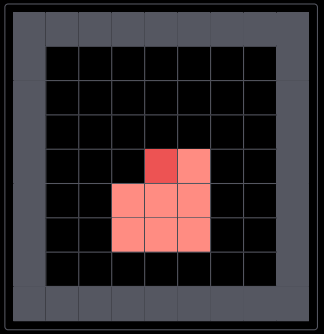
Two Dimensional (2D) Lists

Escape



→ ↑ ↑ ← ← ← ↑ ↑

Dead End



→ ↓ ↓ ← ← ↑ →

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

× selfavoid.py 1/2

```
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
```

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

× selfavoid.py 2/2

```
        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")
```

Converting 2D Lists to 1D Lists and Vice Versa

Converting 2D Lists to 1D Lists and Vice Versa

Converting an $m \times n$ list X into a 1D list Y

- The element $X(i, j)$ maps to the element $Y(k)$, where $k = n \cdot i + j$

Converting 2D Lists to 1D Lists and Vice Versa

Converting an $m \times n$ list X into a 1D list Y

- The element $X(i, j)$ maps to the element $Y(k)$, where $k = n \cdot i + j$

Converting a 1D list Y of size l into an $m \times n$ list X

- The element $Y(k)$ maps to the element $X(i, j)$, where $i = \left\lfloor \frac{k}{n} \right\rfloor$, $j = k \bmod n$, and $m = \frac{l}{n}$

Converting 2D Lists to 1D Lists and Vice Versa

Converting an $m \times n$ list X into a 1D list Y

- The element $X(i, j)$ maps to the element $Y(k)$, where $k = n \cdot i + j$

Converting a 1D list Y of size l into an $m \times n$ list X

- The element $Y(k)$ maps to the element $X(i, j)$, where $i = \left\lfloor \frac{k}{n} \right\rfloor$, $j = k \bmod n$, and $m = \frac{l}{n}$

Example

$$X = \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & \\ \end{array} \\ \begin{bmatrix} A & B & C & D & E \\ F & G & H & I & J \\ K & L & M & N & O \end{bmatrix} \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \end{array}$$

$$Y = \begin{array}{cccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & \\ [A & B & C & D & E & F & G & H & I & J & K & L & M & N & O &] \end{array}$$

Ragged Lists

Ragged Lists

A ragged list is a 2D list in which the rows have unequal number of columns

Ragged Lists

A ragged list is a 2D list in which the rows have unequal number of columns

The number of columns in the `i`th row of a ragged list `x` is obtained as `x[i].length`

Ragged Lists

A ragged list is a 2D list in which the rows have unequal number of columns

The number of columns in the ith row of a ragged list `x` is obtained as `x[i].length`

Example (printing a ragged list)

```
import stdio

pascal = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]

for i in range(len(pascal)):
    for j in range(len(pascal[i])):
        stdio.write(str(pascal[i][j]) + " ")
    stdio.writeln()
```

writes

```
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
>>> _
```

Tuples

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

```
× ~/workspace/ipp
>>> t = 42, 1729, "Hello"
```

Tuples

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

```
× ~/workspace/ipp
>>> t = 42, 1729, "Hello"
>>> _
```

Tuples

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

```
× ~/workspace/ipp
>>> t = 42, 1729, "Hello"
>>> t
```

Tuples

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

```
× ~/workspace/ipp
>>> t = 42, 1729, "Hello"
>>> t
(42, 1729, "Hello")
>>> _
```

Tuples

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

```
× ~/workspace/ipp
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> _
```

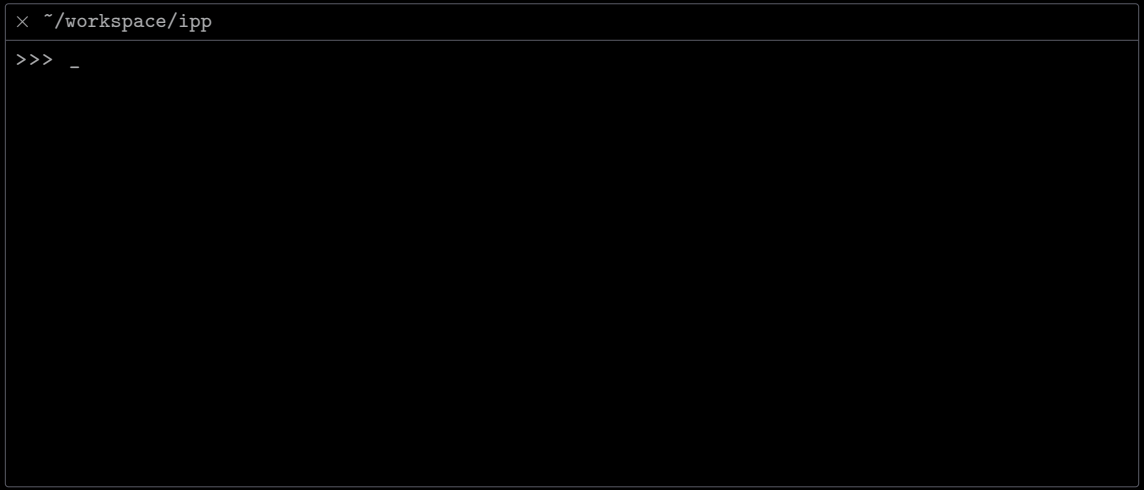
Sets

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



Sets

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

```
× ~/workspace/ipp
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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

```
× ~/workspace/ipp
>>> 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

```
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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
>>> 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

```
× ~/workspace/ipp
>>> 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
>>> 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
>>> 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

× ~/workspace/ipp

```
>>> 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

```
>>> 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

```
>>> 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
>>> 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"}
>>> _
```

Advanced Looping Techniques

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1  import stdio
2
3  for i, v in enumerate(["A", "B", "C"]):
4      stdio.writeln(str(i) + " " + v)
```

line #	i	v

×

```
1
2
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
1		

```
1
2
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1  import stdio
2
3  for i, v in enumerate(["A", "B", "C"]):
4      stdio.writeln(str(i) + " " + v)
```

line #	i	v
3	0	"A"

```
1
2
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
4	0	"A"

```
1 0 A
2
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
3	1	"B"

```
1 0 A
2
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
4	1	"B"

```
1 0 A
2 1 B
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
3	2	"C"

```
1 0 A
2 1 B
3
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
4	2	"C"

```
×
1 0 A
2 1 B
3 2 C
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v
3		

```
0 A
1 B
2 C
```

Advanced Looping Techniques

`enumerate(x)` allows us to loop over the sequence `x` with access to both its elements and their indices

Example

```
1 import stdio
2
3 for i, v in enumerate(["A", "B", "C"]):
4     stdio.writeln(str(i) + " " + v)
```

line #	i	v

```
0 A
1 B
2 C
```

Advanced Looping Techniques

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a

```
1
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
1		

×

```
1
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
3		

×

```
1
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
4		

×

```
1
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1  import stdio
2
3  questions = ["Name", "Age", "Hobby"]
4  answers = ["Alice", "19", "Coding"]
5  for q, a in zip(questions, answers):
6      stdio.writeln(q + "? It is " + a)
```

line #	q	a
5	"Name"	"Alice"

×

```
1
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
6	"Name"	"Alice"

×

```
1 Name? It is Alice.
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
5	"Age"	"19"

×

```
1 Name? It is Alice.
2
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, ... at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
6	"Age"	"19"

×

```
1 Name? It is Alice.
2 Age? It is 19.
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1  import stdio
2
3  questions = ["Name", "Age", "Hobby"]
4  answers = ["Alice", "19", "Coding"]
5  for q, a in zip(questions, answers):
6      stdio.writeln(q + "? It is " + a)
```

line #	q	a
5	"Hobby"	"Coding"

×

```
1  Name? It is Alice.
2  Age? It is 19.
3
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
6	"Hobby"	"Coding"

×

```
1 Name? It is Alice.
2 Age? It is 19.
3 Hobby? It is Coding.
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a
5		

×

```
1 Name? It is Alice.
2 Age? It is 19.
3 Hobby? It is Coding.
```

Advanced Looping Techniques

`zip(x, y, ...)` allows us to loop over two or more equal-length sequences `x`, `y`, `...` at the same time

Example

```
1 import stdio
2
3 questions = ["Name", "Age", "Hobby"]
4 answers = ["Alice", "19", "Coding"]
5 for q, a in zip(questions, answers):
6     stdio.writeln(q + "? It is " + a)
```

line #	q	a

×

```
1 Name? It is Alice.
2 Age? It is 19.
3 Hobby? It is Coding.
```

Advanced Looping Techniques

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v

```
1
2
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
1		

```
1
2
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
3	["A", "B", "C"]	

```
1
2
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
4	["A", "B", "C"]	"C"

```
1
2
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1 import stdio
2
3 x = ["A", "B", "C"]
4 for v in reversed(x):
5     stdio.writeln(v)
```

line #	x	v
5	["A", "B", "C"]	"C"

```
×
1 C
2
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
4	["A", "B", "C"]	"B"

```
1  C
2
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
5	["A", "B", "C"]	"B"

```
1  C
2  B
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
4	["A", "B", "C"]	"A"

```
1  C
2  B
3
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
5	["A", "B", "C"]	"A"

×

```
1  C
2  B
3  A
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v
4	["A", "B", "C"]	

×

```
1  C
2  B
3  A
```

Advanced Looping Techniques

`reversed(x)` allows us to loop over the sequence `x` in reverse

Example

```
1  import stdio
2
3  x = ["A", "B", "C"]
4  for v in reversed(x):
5      stdio.writeln(v)
```

line #	x	v

```
1  C
2  B
3  A
```

Advanced Looping Techniques

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1 import stdio
2
3 x = ["B", "A", "C"]
4 for v in sorted(x):
5     stdio.writeln(v)
```

line #	x	v

```
1
2
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
1		

```
1
2
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
3	["B", "A", "C"]	

```
1
2
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
4	["B", "A", "C"]	"A"

```
1
2
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
5	["B", "A", "C"]	"A"

```
1  A
2
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
4	["B", "A", "C"]	"B"

```
1  A
2
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
5	["B", "A", "C"]	"B"

```
1  A
2  B
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
4	["B", "A", "C"]	"C"

```
1  A
2  B
3
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
5	["B", "A", "C"]	"C"

```
1  A
2  B
3  C
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
4	["B", "A", "C"]	

```
1  A
2  B
3  C
```

Advanced Looping Techniques

`sorted(x)` allows us to loop over the sequence `x` in sorted order

Example

```
1  import stdio
2
3  x = ["B", "A", "C"]
4  for v in sorted(x):
5      stdio.writeln(v)
```

line #	x	v
4		

```
1  A
2  B
3  C
```