

Introduction to Programming in Python

Imperative Programming: Collection Data Types

Outline

- ➊ One Dimensional (1D) Lists
- ➋ Two Dimensional (2D) Lists
- ➌ Converting 2D Lists to 1D Lists and Vice Versa
- ➍ Ragged Lists
- ➎ Tuples
- ➏ Sets
- ➐ Advanced Looping Techniques

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

```
1 <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

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

Example

```
1 suits = ["Clubs", "Diamonds", "Hearts", "Spades"]  
2 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

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


One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i
1	[]	

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i
2	[]	0

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i
3	[0]	0

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i
2	[0]	1

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i
3	[0, 0]	1

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

line #	a	i
2	[0, 0]	2

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

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

One Dimensional (1D) Lists

Appending to a 1D list

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

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

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

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

One Dimensional (1D) Lists

Appending to a 1D list

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

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

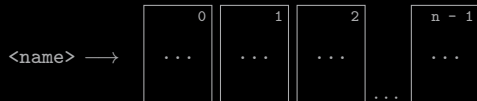
```
1 a = []  
2 for i in range(3):  
3     a += [0]
```

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 i th element in a list x is accessed as $x[i]$, where $0 \leq i < \text{len}(x)$

One Dimensional (1D) Lists

The i th element in a list x is accessed as $x[i]$, where $0 \leq i < \text{len}(x)$

The i th element in a list x is assigned a value as

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

One Dimensional (1D) Lists

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)
```

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)

```
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)
```

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
2
3
4
5
6

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

1 \$ _

2

3

4

5

6

7

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

```
1 $ python3 sample.py 6 16
```

2

3

4

5

6

7

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

```
1 $ python3 sample.py 6 16
```

```
2 11 10 12 13 6 8
```

```
3 $ _
```

```
4
```

```
5
```

```
6
```

```
7
```


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

```
1 $ python3 sample.py 6 16
```

```
2 11 10 12 13 6 8
```

```
3 $ python3 sample.py 10 1000
```

```
4
```

```
5
```

```
6
```

```
7
```

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

```
1 $ python3 sample.py 6 16
```

```
2 11 10 12 13 6 8
```

```
3 $ python3 sample.py 10 1000
```

```
4 21 432 270 287 166 484 437 675 78 213
```

```
5 $ _
```

```
6
```

```
7
```

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

```
1 $ python3 sample.py 6 16
```

```
2 11 10 12 13 6 8
```

```
3 $ python3 sample.py 10 1000
```

```
4 21 432 270 287 166 484 437 675 78 213
```

```
5 $ python3 sample.py 20 20
```

```
6
```

```
7
```

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

```
1 $ python3 sample.py 6 16
```

```
2 11 10 12 13 6 8
```

```
3 $ python3 sample.py 10 1000
```

```
4 21 432 270 287 166 484 437 675 78 213
```

```
5 $ python3 sample.py 20 20
```

```
6 9 0 15 13 4 8 11 17 3 18 16 5 7 19 14 12 2 1 10 6
```

```
7 $ _
```

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

```
1 import stdarray
2 import stdio
3 import stdrandom
4 import sys
5
6 m = int(sys.argv[1])
7 n = int(sys.argv[2])
8 perm = stdarray.create1D(n, 0)
9 for i in range(n):
10     perm[i] = i
11 for i in range(m):
12     r = stdrandom.uniformInt(i, n)
13     temp = perm[r]
14     perm[r] = perm[i]
15     perm[i] = temp
16 for i in range(m):
17     stdio.write(str(perm[i]) + " ")
18 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

1 \$ _

2

3

4

5

6

7

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

```
1 $ python3 couponcollector.py 1000
```

2

3

4

5

6

7

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

```
1 $ python3 couponcollector.py 1000
```

```
2 8317
```

```
3 $ _
```

```
4
```

```
5
```

```
6
```

```
7
```

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

```
1 $ python3 couponcollector.py 1000
```

```
2 8317
```

```
3 $ python3 couponcollector.py 1000
```

```
4
```

```
5
```

```
6
```

```
7
```


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

```
1 $ python3 couponcollector.py 1000
```

```
2 8317
```

```
3 $ python3 couponcollector.py 1000
```

```
4 7867
```

```
5 $ _
```

```
6
```

```
7
```

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

```
1 $ python3 couponcollector.py 1000
2 8317
3 $ python3 couponcollector.py 1000
4 7867
5 $ 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

```
1 $ python3 couponcollector.py 1000
2 8317
3 $ python3 couponcollector.py 1000
4 7867
5 $ python3 couponcollector.py 1000000
6 15942756
7 $ _
```

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

```
1 import stdarray
2 import stdio
3 import stdrandom
4 import sys
5
6 n = int(sys.argv[1])
7 count = 0
8 collectedCount = 0
9 isCollected = stdarray.create1D(n, False)
10 while collectedCount < n:
11     value = stdrandom.uniformInt(0, n)
12     count += 1
13     if not isCollected[value]:
14         collectedCount += 1
15         isCollected[value] = True
16 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

1 \$ _

2

3

4

5

6

7

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

```
1 $ python3 primesieve.py 10
```

2

3

4

5

6

7

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

```
1 $ python3 primesieve.py 10
```

```
2 4
```

```
3 $ _
```

```
4
```

```
5
```

```
6
```

```
7
```

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

```
1 $ python3 primesieve.py 10
```

```
2 4
```

```
3 $ python3 primesieve.py 100
```

```
4
```

```
5
```

```
6
```

```
7
```

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

```
1 $ python3 primesieve.py 10
```

```
2 4
```

```
3 $ python3 primesieve.py 100
```

```
4 25
```

```
5 $ _
```

```
6
```

```
7
```

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

```
1 $ python3 primesieve.py 10
```

```
2 4
```

```
3 $ python3 primesieve.py 100
```

```
4 25
```

```
5 $ python3 primesieve.py 1000
```

```
6
```

```
7
```

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

```
1 $ python3 primesieve.py 10
```

```
2 4
```

```
3 $ python3 primesieve.py 100
```

```
4 25
```

```
5 $ python3 primesieve.py 1000
```

```
6 168
```

```
7 $ _
```

One Dimensional (1D) Lists

• `list` (built-in)

• `array` (NumPy)

• `matrix` (SciPy)

• `ndarray` (NumPy)

• `matrix` (SciPy)

• `array` (NumPy)

• `matrix` (SciPy)

• `array` (NumPy)

• `matrix` (SciPy)

• `array` (NumPy)

• `matrix` (SciPy)

One Dimensional (1D) Lists

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

```
1 import stdio
2
3 x = [1, 3, 7]
4 y = x
5 x[1] = 42
6
7 stdio.writeln(x)
8 stdio.writeln(y)
```

writes

```
1 [1, 42, 7]
2 [1, 42, 7]
```

One Dimensional (1D) Lists

One Dimensional (1D) Lists

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

- Method 1 (using a loop)

```
1 y = []  
2 for v in x:  
3     y += [v]
```


One Dimensional (1D) Lists

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

- Method 1 (using a loop)

```
1 y = []  
2 for v in x:  
3     y += [v]
```

- Method 2 (using the slicing operator)

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

One Dimensional (1D) Lists

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

- Method 1 (using a loop)

```
1 y = []  
2 for v in x:  
3     y += [v]
```

- Method 2 (using the slicing operator)

```
1 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

```
1 import stdio
2
3 s = "Hello, World!"
4 for c in s[7 : len(s) - 1]:
5     stdio.write(c)
6 stdio.writeln()
```

writes

```
1 World
```

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

Creating a 2D list

```
1 <name> = [[<expression>, <expression>, ..., <expression>],  
2          [<expression>, <expression>, ..., <expression>],  
3          ...  
4          [<expression>, <expression>, ..., <expression>]]
```

Two Dimensional (2D) Lists

Creating a 2D list

```
1 <name> = [[<expression>, <expression>, ..., <expression>],  
2          [<expression>, <expression>, ..., <expression>],  
3          ...  
4          [<expression>, <expression>, ..., <expression>]]
```

Example

```
1 identity = [[1, 0, 0],  
2             [0, 1, 0],  
3             [0, 0, 1]]  
4  
5 pascal = [[1, 0, 0, 0, 0],  
6           [1, 1, 0, 0, 0],  
7           [1, 2, 1, 0, 0],  
8           [1, 3, 3, 1, 0],  
9           [1, 4, 6, 4, 1]]
```


Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

Appending to a 2D list

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

```
1 import ndarray
2
3 a = []
4 for i in range(3):
5     a += [ndarray.create1D(2, 0)]
```

line #	a	i

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

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

line #	a	i
1		

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

```
1 import ndarray
2
3 a = []
4 for i in range(3):
5     a += [ndarray.create1D(2, 0)]
```

line #	a	i
3	[]	

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

```
1 import ndarray
2
3 a = []
4 for i in range(3):
5     a += [ndarray.create1D(2, 0)]
```

line #	a	i
4	[]	0

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

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

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

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

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

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

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

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

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

```
1 import ndarray
2
3 a = []
4 for i in range(3):
5     a += [ndarray.create1D(2, 0)]
```

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

```
1 import ndarray
2
3 a = []
4 for i in range(3):
5     a += [ndarray.create1D(2, 0)]
```

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

Two Dimensional (2D) Lists

Appending to a 2D list

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

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

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

line #	a	i

Two Dimensional (2D) Lists

Two dimensional lists are:

lists of lists

lists of tuples

lists of sets

lists of dictionaries

lists of objects

lists of lists

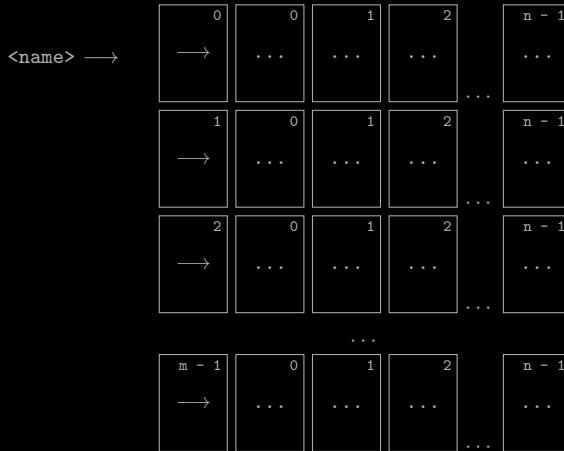
lists of tuples

lists of sets

lists of dictionaries

lists of objects

Two Dimensional (2D) Lists



Two Dimensional (2D) Lists

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list2 = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
list3 = [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
list4 = [31, 32, 33, 34, 35, 36, 37, 38, 39, 40]
list5 = [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
list6 = [51, 52, 53, 54, 55, 56, 57, 58, 59, 60]
list7 = [61, 62, 63, 64, 65, 66, 67, 68, 69, 70]
list8 = [71, 72, 73, 74, 75, 76, 77, 78, 79, 80]
list9 = [81, 82, 83, 84, 85, 86, 87, 88, 89, 90]
list10 = [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

# Create a 2D list (matrix)
matrix = [list1, list2, list3, list4, list5, list6, list7, list8, list9, list10]

# Accessing elements in a 2D list
# Accessing the first row
first_row = matrix[0]
print(first_row)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Accessing the second row
second_row = matrix[1]
print(second_row)  # Output: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

# Accessing the third row
third_row = matrix[2]
print(third_row)  # Output: [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

# Accessing the fourth row
fourth_row = matrix[3]
print(fourth_row)  # Output: [31, 32, 33, 34, 35, 36, 37, 38, 39, 40]

# Accessing the fifth row
fifth_row = matrix[4]
print(fifth_row)  # Output: [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]

# Accessing the sixth row
sixth_row = matrix[5]
print(sixth_row)  # Output: [51, 52, 53, 54, 55, 56, 57, 58, 59, 60]

# Accessing the seventh row
seventh_row = matrix[6]
print(seventh_row)  # Output: [61, 62, 63, 64, 65, 66, 67, 68, 69, 70]

# Accessing the eighth row
eighth_row = matrix[7]
print(eighth_row)  # Output: [71, 72, 73, 74, 75, 76, 77, 78, 79, 80]

# Accessing the ninth row
ninth_row = matrix[8]
print(ninth_row)  # Output: [81, 82, 83, 84, 85, 86, 87, 88, 89, 90]

# Accessing the tenth row
tenth_row = matrix[9]
print(tenth_row)  # Output: [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

# Accessing a specific element in a 2D list
# Accessing the element at row 2, column 3
element = matrix[2][2]
print(element)  # Output: 23

# Accessing the element at row 5, column 7
element = matrix[5][6]
print(element)  # Output: 57

# Accessing the element at row 8, column 1
element = matrix[8][0]
print(element)  # Output: 81

# Accessing the element at row 10, column 10
element = matrix[9][9]
print(element)  # Output: 100
```


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 lists are:

• Lists of lists

• Lists of tuples

• Lists of sets

• Lists of dictionaries

• Lists of objects

• Lists of lists

• Lists of tuples

• Lists of sets

• Lists of dictionaries

• Lists of objects

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 $0 \leq i < m$ and $0 \leq j < n$

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 $0 \leq i < m$ and $0 \leq j < n$

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

```
1 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

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list2 = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

# Create a 2D list
list3 = [list1, list2]

# Print the 2D list
print(list3)
```

The output of the code is:

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]]
```

A 2D list is a list of lists. It is a list where each element is itself a list. In the example above, `list3` is a 2D list containing two 1D lists, `list1` and `list2`. The output shows that `list3` is a list containing two elements, each of which is a list of 10 integers.

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

1 \$ _

2

3

4

5

6

7

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

```
1 $ python3 selfavoid.py 20 1000
```

2
3
4
5
6
7

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

```
1 $ python3 selfavoid.py 20 1000
```

```
2 32% dead ends
```

```
3 $ _
```

```
4
```

```
5
```

```
6
```

```
7
```

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

```
1 $ python3 selfavoid.py 20 1000
```

```
2 32% dead ends
```

```
3 $ python3 selfavoid.py 40 1000
```

```
4
```

```
5
```

```
6
```

```
7
```

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

```
1 $ python3 selfavoid.py 20 1000
```

```
2 32% dead ends
```

```
3 $ python3 selfavoid.py 40 1000
```

```
4 75% dead ends
```

```
5 $ _
```

```
6
```

```
7
```


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

```
1 $ python3 selfavoid.py 20 1000
```

```
2 32% dead ends
```

```
3 $ python3 selfavoid.py 40 1000
```

```
4 75% dead ends
```

```
5 $ python3 selfavoid.py 80 1000
```

```
6
```

```
7
```

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

```
1 $ python3 selfavoid.py 20 1000
```

```
2 32% dead ends
```

```
3 $ python3 selfavoid.py 40 1000
```

```
4 75% dead ends
```

```
5 $ python3 selfavoid.py 80 1000
```

```
6 98% dead ends
```

```
7 $ _
```

Two Dimensional (2D) Lists

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list2 = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
list3 = [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
list4 = [31, 32, 33, 34, 35, 36, 37, 38, 39, 40]
list5 = [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
list6 = [51, 52, 53, 54, 55, 56, 57, 58, 59, 60]
list7 = [61, 62, 63, 64, 65, 66, 67, 68, 69, 70]
list8 = [71, 72, 73, 74, 75, 76, 77, 78, 79, 80]
list9 = [81, 82, 83, 84, 85, 86, 87, 88, 89, 90]
list10 = [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

# Create a 2D list (matrix)
matrix = [list1, list2, list3, list4, list5, list6, list7, list8, list9, list10]

# Accessing elements in a 2D list
# Accessing the first row
first_row = matrix[0]
print(first_row)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Accessing the second row
second_row = matrix[1]
print(second_row)  # Output: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

# Accessing the third row
third_row = matrix[2]
print(third_row)  # Output: [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

# Accessing the fourth row
fourth_row = matrix[3]
print(fourth_row)  # Output: [31, 32, 33, 34, 35, 36, 37, 38, 39, 40]

# Accessing the fifth row
fifth_row = matrix[4]
print(fifth_row)  # Output: [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]

# Accessing the sixth row
sixth_row = matrix[5]
print(sixth_row)  # Output: [51, 52, 53, 54, 55, 56, 57, 58, 59, 60]

# Accessing the seventh row
seventh_row = matrix[6]
print(seventh_row)  # Output: [61, 62, 63, 64, 65, 66, 67, 68, 69, 70]

# Accessing the eighth row
eighth_row = matrix[7]
print(eighth_row)  # Output: [71, 72, 73, 74, 75, 76, 77, 78, 79, 80]

# Accessing the ninth row
ninth_row = matrix[8]
print(ninth_row)  # Output: [81, 82, 83, 84, 85, 86, 87, 88, 89, 90]

# Accessing the tenth row
tenth_row = matrix[9]
print(tenth_row)  # Output: [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

# Accessing a specific element
# Accessing the element at row 3, column 4
element = matrix[3][4]
print(element)  # Output: 35

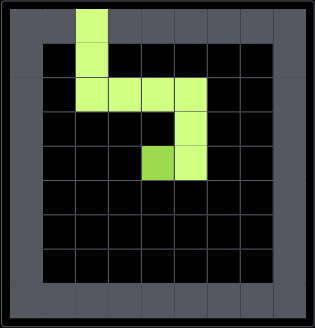
# Accessing the element at row 7, column 8
element = matrix[7][8]
print(element)  # Output: 79

# Accessing the element at row 9, column 1
element = matrix[9][0]
print(element)  # Output: 91

# Accessing the element at row 10, column 10
element = matrix[10][9]
print(element)  # Output: 100
```

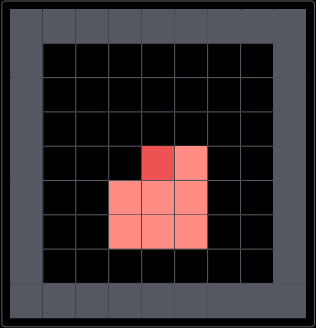
Two Dimensional (2D) Lists

Escape



→ ↑ ↑ ← ← ← ↑ ↑

Dead End



→ ↓ ↓ ← ← ↑ →

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

× selfavoid.py

1/2

```
1 import stdarray
2 import stdio
3 import stdrandom
4 import sys
5
6 n = int(sys.argv[1])
7 trials = int(sys.argv[2])
8 deadEnds = 0
9 for t in range(trials):
10     a = stdarray.create2D(n, n, False)
11     x = n // 2
12     y = n // 2
13     while x > 0 and x < n - 1 and y > 0 and y < n - 1:
14         a[x][y] = True
15         if a[x - 1][y] and a[x + 1][y] and a[x][y - 1] and a[x][y + 1]:
16             deadEnds += 1
17             break
18         r = stdrandom.uniformInt(1, 5)
19         if r == 1 and not a[x + 1][y]:
20             x += 1
```

Two Dimensional (2D) Lists

Two Dimensional (2D) Lists

× selfavoid.py

2/2

```
21         elif r == 2 and not a[x - 1][y]:
22             x -= 1
23         elif r == 3 and not a[x][y + 1]:
24             y += 1
25         elif r == 4 and not a[x][y - 1]:
26             y -= 1
27     stdout.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

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

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

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 `i`th row of a ragged list `x` is obtained as `x[i].length`

Example (printing a ragged list)

```
1 import stdio
2
3 pascal = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
4
5 for i in range(len(pascal)):
6     for j in range(len(pascal[i])):
7         stdio.write(str(pascal[i][j]) + " ")
8     stdio.writeln()
```

writes

```
1 1
2 1 1
3 1 2 1
4 1 3 3 1
5 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

```
1 >>> _
```

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> _
```

3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> _
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"  
2 >>> t  
3 (42, 1729, "Hello")  
4 >>> 1729 in t  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18
```


Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"  
2 >>> t  
3 (42, 1729, "Hello")  
4 >>> 1729 in t  
5 True  
6 >>> _
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"  
2 >>> t  
3 (42, 1729, "Hello")  
4 >>> 1729 in t  
5 True  
6 >>> t[1]
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
2 >>> t
3 (42, 1729, "Hello")
4 >>> 1729 in t
5 True
6 >>> t[1]
7 1729
8 >>> _
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
2 >>> t
3 (42, 1729, "Hello")
4 >>> 1729 in t
5 True
6 >>> t[1]
7 1729
8 >>> t[2] = "Hello, World"
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> _
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> _
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> len(empty)
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```


Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> len(empty)
```

```
14 0
```

```
15 >>> _
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> len(empty)
```

```
14 0
```

```
15 >>> singleton = "Hello",
```

```
16
```

```
17
```

```
18
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> len(empty)
```

```
14 0
```

```
15 >>> singleton = "Hello",
```

```
16 >>> _
```

```
17
```

```
18
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> len(empty)
```

```
14 0
```

```
15 >>> singleton = "Hello",
```

```
16 >>> len(singleton)
```

```
17
```

```
18
```

Tuples

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

× ~/workspace/ipp

```
1 >>> t = 42, 1729, "Hello"
```

```
2 >>> t
```

```
3 (42, 1729, "Hello")
```

```
4 >>> 1729 in t
```

```
5 True
```

```
6 >>> t[1]
```

```
7 1729
```

```
8 >>> t[2] = "Hello, World"
```

```
9 Traceback (most recent call last):
```

```
10   File "<stdin>", line 1, in <module>
```

```
11 TypeError: 'tuple' object does not support item assignment
```

```
12 >>> empty = ()
```

```
13 >>> len(empty)
```

```
14 0
```

```
15 >>> singleton = "Hello",
```

```
16 >>> len(singleton)
```

```
17 1
```

```
18 >>> _
```

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

```
1 >>> _
```

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
```

```
2 >>> _
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]  
2 >>> fruit = set(basket)  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> _
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> _
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> _
```


Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8
9
10
11
12
13
14
15
16
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> _
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9
10
11
12
13
14
15
16
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> _
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> _
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> a | b
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> a | b
12 {"l", "c", "d", "z", "a", "r", "m", "b"}
13 >>> _
```


Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> a | b
12 {"l", "c", "d", "z", "a", "r", "m", "b"}
13 >>> a & b
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> a | b
12 {"l", "c", "d", "z", "a", "r", "m", "b"}
13 >>> a & b
14 {"c", "a"}
15 >>> _
16
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> a | b
12 {"l", "c", "d", "z", "a", "r", "m", "b"}
13 >>> a & b
14 {"c", "a"}
15 >>> a ^ b
16
17
```

Sets

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

× ~/workspace/ipp

```
1 >>> basket = ["orange", "apple", "pear", "orange", "banana", "apple"]
2 >>> fruit = set(basket)
3 >>> fruit
4 {"banana", "pear", "orange", "apple"}
5 >>> "orange" in fruit
6 True
7 >>> a = set("abracadabra")
8 >>> b = set("alacazam")
9 >>> a - b
10 {"b", "d", "r"}
11 >>> a | b
12 {"l", "c", "d", "z", "a", "r", "m", "b"}
13 >>> a & b
14 {"c", "a"}
15 >>> a ^ b
16 {"l", "r", "d", "m", "b", "z"}
17 >>> _
```

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		

	×
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

	×
1	0 A
2	1 B
3	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 ×
2 Name? It is Alice.
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