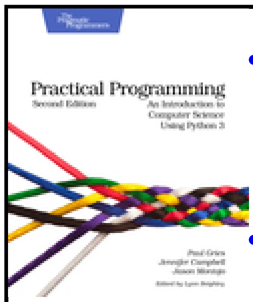
Type `list`



- Chapter 8.1 Storing and Accessing Data in Lists
- Chapter 8.3 Operations on Lists
- Chapter 8.4 Slicing Lists

Optional reading

Overview

Our programs will often work with collections of data. One way to store these collections of data is using Python's type `list`.

The general form of a list is:

```
[expr1, expr2, ..., exprN]
```

For example, here is a list of three grades:

```
grades = [80, 90, 70]
```

List Operations

Like strings, lists can be indexed:

```
>>> grades[0]
80
>>> grades[1]
90
>>> grades[2]
70
```

Lists can also be sliced, using the same notation as for strings:

```
>>> grades[0:2]
[80, 90]
```

The `in` operator can also be applied to check whether a value is an item in a list.

```
>>> 90 in grades
True
>>> 60 in grades
False
```

Several of Python's built-in functions can be applied to lists, including:

- `len(list)`: return the length of `list`.
- `min(list)`: return the smallest element in `list`.
- `max(list)`: return the largest element in `list`.
- `sum(list)`: return the sum of elements of `list` (where list items must be numeric).

For example, here are some calls to those built-in functions:

```
>>> len(grades)
3
>>> min(grades)
70
>>> max(grades)
90
>>> sum(grades)
```

```
240
```

Types of list elements

Lists elements may be of any type. For example, here is a `list of str`:

```
subjects = ['bio', 'cs', 'math', 'history']
```

Lists can also contain elements of more than one type. For example, a street address can be represented by a `list of [int, str]`:

```
street_address = [10, 'Main Street']
```

`for` loops over `list`

Similar to looping over the characters of a string, it is possible to iterate over the elements of a list. For example:

```
>>> for grade in grades:
    print(grade)
80
90
70
```

The general form of a `for` loop over a `list` is:

```
for variable in list:
    body
```

Jennifer Campbell • Paul Gries
University of Toronto