Chapter 3 Python Object Types

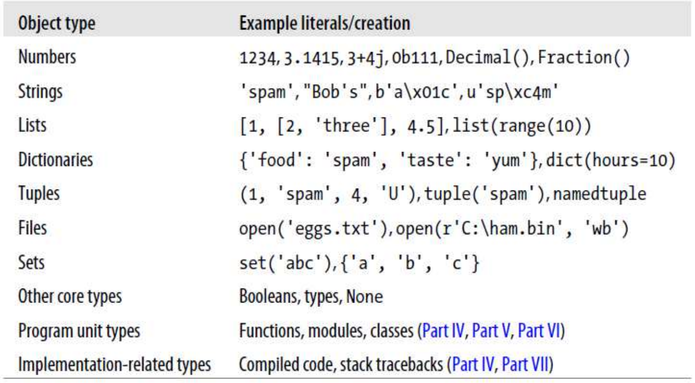
The Python Conceptual Hierarchy

- Python programs can be decomposed into modules, statements, expressions, and objects, as follows:
    1. Programs are composed of modules.
    2. Modules contain statements.
    3. Statements contain expressions.
    4. *Expressions create and process objects*.

Why Use Built-in Types?

- Built-in objects make programs easy to write.
- Built-in objects are components of extensions.
- Built-in objects are often more efficient than custom data structures.
- Built-in objects are a standard part of the language.

Python’s Core Data Types

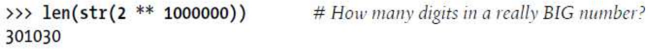


Object type	Example literals/creation
Numbers	`1234`, `3.1415`, `3+4j`, `0b111`, `Decimal()`, `Fraction()`
Strings	`'spam'`, `"Bob's"`, `b'a\x01c'`, `u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5]`, `list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}`, `dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U')`, `tuple('spam')`, `namedtuple`
Files	`open('eggs.txt')`, `open(r'C:\ham.bin', 'wb')`
Sets	`set('abc')`, `{'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes (Part IV, Part V, Part VI)
Implementation-related types	Compiled code, stack tracebacks (Part IV, Part VII)

Numbers

- ***integers*** that have no fractional part, ***floating-point*** numbers that do, and more exotic types—***complex*** numbers with imaginary parts, ***decimals*** with fixed precision, ***rationals*** with numerator and denominator, and full-featured *sets*.
- Numbers in Python support the normal mathematical operations.

❖ You can, for instance, compute 2 to the power 1,000,000 as an integer in Python, but you probably shouldn’t try to print the result—with more than 300,000 digits, you may be waiting awhile!



```
>>> len(str(2 ** 1000000))        # How many digits in a really BIG number?
301030
```

Strings

- Strings are used to record both textual information as well as arbitrary collections of bytes.
- ***sequence***—a positionally ordered collection of other objects.
- their items are stored and fetched by their relative positions.
- Strings are sequences of one-character strings.

Sequence Operations

- strings support operations that assume a positional ordering among items.
- we can verify string’s length with the built-in **len** function and fetch its components with ***indexing*** expressions.

```
S = ‘test’
len(S) => 4
S[0] => ‘S’
```

- **indexes** are coded as **offsets** from the **front**, and so start from 0
- A variable is created when you assign it a value.
- we can also index **backward**, from the end—**positive indexes** count from the **left**, and **negative indexes** count back from the **right.**

```
S[-2] => ‘s’
```

- we can use an ***arbitrary expression*** in the square brackets.
- strings also support *concatenation* with the plus sign (joining two strings into a new string) and *repetition* (making a new string by repeating another):

```
S + ‘xyz’ => ‘testxyz’
S => ‘test’
S * 3 => ‘testtesttest’
```

Immutability

- Every string operation is defined to produce a new string as its result, because strings are ***immutable*** in Python.
- you can never overwrite the values of immutable objects.

```
S = ‘a’ + S[1:] => 'aest'    # we can do this
S[1] = ‘a’          #this will throw error.
```

Type-Specific Methods

- Strings also support an advanced substitution operation known as ***formatting***.

```
>>> '{}, eggs, and {}'.format('spam', 'SPAM!') # Numbers optional (2.7+, 3.1+)
'spam, eggs, and SPAM!'
```

- >>> dir(S)
- >>> help(S.replace)

Other Ways to Code Strings

- Python allows strings to be enclosed in ***single*** or ***double*** quote characters
- multiline string literals enclosed in *triple* quotes (single or double)

Unicode Strings

- Python’s strings also come with full *Unicode* support required for processing text in internationalized character sets.

Lists

- Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size.
- They are also *mutable.*
- lists can be modified in place by assignment to offsets as well as a variety of list method calls.

Sequence Operations

```
L = [123, ‘test’, 45.65]    #list of 3 different elements
```

len(L) => 3

len[0] => 123

L[:-1] => [123, ‘test’]

L + [4, 5, 6] => [123, test, 1.23, 4, 5, 6]

Type-Specific Operations – methods in list

- Has no fixed type.
- List has no fixed size.
- List can grow and shrink on demand.
  - append()
  - pop()

Bounds Checking

- Indexing off the end of a list is always a mistake, but so is assigning off the end

Nesting

- list support arbitrary *nesting.*
- We can nest them in any combination, and as deeply as we like.

Comprehensions

l = [5,6,8,6,2,5,4]

[x*2 for x in l] => [10, 12, 16, 12, 4, 10, 8]

[x*2 for x in l if x%2 == 0] => [12, 16, 12, 4, 8]

- List comprehensions make new lists of results, but they can be used to iterate over any *iterable* object

Dictionaries

- *mappings*.
- Mappings are also collections of other objects, but they store objects by ***key*** instead of by relative position.
- mappings don’t maintain any reliable left-to-right order
- they simply map keys to associated values
- also *mutable*

Mapping Operations

- dictionaries are coded in curly braces and consist of a series of “key: value” pairs.

  D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
- can index this dictionary by key to fetch and change the keys’ associated values.

  D['food'] => 'Spam'

  D['quantity'] += 1 => {'color': 'pink', 'food': 'Spam', 'quantity': 5}
- indexing a dictionary by key is often the fastest way to code a search in Python.
- we can also make dictionaries by passing to the dict type name either *keyword arguments* (a special *name*=*value* syntax in function calls), or the result of *zipping* together sequences of keys and values obtained at runtime.

- #keywords

  bob1 = dict(name='Bob', job='dev', age=40) => {'age': 40, 'name': 'Bob', 'job': 'dev'}
- #Zipping

bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40])) => {'job': 'dev', 'name': 'Bob', 'age': 40}

Missing Keys: if Tests

D = {'a': 1, 'b': 2, 'c': 3} => {'a': 1, 'c': 3, 'b': 2}

D['e'] = 99

D => {'a': 1, 'c': 3, 'b': 2, 'e': 99}

D['f'] => #error

Iteration and Optimization

squares = [x ** 2 for x in [1, 2, 3, 4, 5]]

Tuples

- roughly like a **list that cannot be changed**—tuples are ***sequences***, like lists, but they are ***immutable***, like strings.

T = (1, 2, 3, 4)

T + (5, 6) => (1, 2, 3, 4, 5, 6) *# Concatenation*

T[0] => 1 *# Indexing, slicing, and more*

- index(), count()
- The primary distinction for tuples is that they cannot be changed once created.

T = (2) + T[1:] => TypeError: unsupported operand type(s) for +: 'int' and 'tuple'

T = (2,) + T[1:] => (2, 2, 3, 4, 5)

Why Tuples?

- tuples provide a sort of integrity constraint that is convenient in large programs.

Files

- File objects are Python code’s main interface to external files on your computer.
- to create a file object, you call the built-in open function, passing in an external filename and an optional processing mode as strings.

f = open('data.txt', 'w') # Make a new file in output mode ('w' is write)

f.write('Hello\n')

for line in open('data.txt'): print(line)

Question:

1. Name four of Python’s core data types.
2. Why are they called “core” data types?
3. What does “immutable” mean, and which three of Python’s core types are considered immutable?
4. What does “sequence” mean, and which three types fall into that category?
5. What does “mapping” mean, and which core type is a mapping?