Python3

Встроенные объекты

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

Numbers

```
>>> 123 + 222 # Integer addition
345
>>> 1.5 * 4 # Floating-point multiplication
6.0
>>> 2 ** 100 # 2 to the power 100, again
1267650600228229401496703205376
```

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

start experimenting with floating-point
numbers, you’re likely to stumble across something that may look a bit odd at first
glance:

```
>>> 3.1415 * 2 # repr: as code (Pythons < 2.7 and 3.1)
6.2830000000000004
>>> print(3.1415 * 2) # str: user-friendly
6.283
```

```
>>> import math
>>> math.pi
```

```
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

```
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
1
```

String

```
>>> S = 'Spam' # Make a 4-character string, and assign it to a name
>>> len(S) # Length
4
>>> S[0] # The first item in S, indexing by zero-based position
'S'
>>> S[1] # The second item from the left
'p'
```

In Python, we can also index backward, from the end—positive indexes count from the left, and negative indexes count back from the right:

```
>>> S[-1] # The last item from the end in S
'm'
>>> S[-2] # The second-to-last item from the end
'a'
```

Formally, a negative index is simply added to the string’s length, so the following two operations are equivalent (though the first is easier to code and less easy to get wrong):

```
>>> S[-1] # The last item in S
'm'
>>> S[len(S)-1] # Negative indexing, the hard way
'm'
```

In addition to simple positional indexing, sequences also support a more general form of indexing known as slicing, which is a way to extract an entire section (slice) in a single step. For example:

```
>>> S # A 4-character string
'Spam'
>>> S[1:3] # Slice of S from offsets 1 through 2 (not 3)
'pa'

>>> S[1:] # Everything past the first (1:len(S))
'pam'
>>> S # S itself hasn't changed
'Spam'
>>> S[0:3] # Everything but the last
'Spa'
>>> S[:3] # Same as S[0:3]
'Spa'
>>> S[:-1] # Everything but the last again, but simpler (0:-1)
'Spa'
>>> S[:] # All of S as a top-level copy (0:len(S))
'Spam'

>>> S + 'xyz' # Concatenation
'Spamxyz'
>>> S # S is unchanged
'Spam'
>>> S * 8 # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Immutability

Also notice in the prior examples that we were not changing the original string with any of the operations we ran on it. Every string operation is defined to produce a new string as its result, because strings are immutable in Python—they cannot be changed in place after they are created. In other words, you can never overwrite the values of immutable objects. For example, you can’t change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name. Because Python cleans up old objects as you go (as you’ll see later), this isn’t as inefficient as it may sound:

```
>>> S
'Spam'
>>> S[0] = 'z' # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment
>>> S = 'z' + S[1:] # But we can run expressions to make new objects
>>> S
'zpam'
```

Every object in Python is classified as either *immutable (unchangeable)* or not. In terms of the core types, *numbers*, *strings*, and *tuples* are immutable; *lists*, *dictionaries*, and *sets* are not—they can be changed in place freely, as can most new objects you’ll code with classes.

Strictly speaking, you can change text-based data in place if you either expand it into a list of individual characters and join it back together with nothing between, or use the newer bytearray type available in Pythons 2.6, 3.0, and later:

```
>>> S = 'shrubbery'
>>> L = list(S) # Expand to a list: [...]
>>> L
['s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'y']
>>> L[1] = 'c' # Change it in place
>>> ''.join(L) # Join with empty delimiter
'scrubbery'
>>> B = bytearray(b'spam') # A bytes/list hybrid (ahead)
>>> B.extend(b'eggs') # 'b' needed in 3.X, not 2.X
>>> B # B[i] = ord(c) works here too
```

```
bytearray(b'spameggs')
>>> B.decode() # Translate to normal string
'spameggs'
```

Type-Specific Methods

For example, the string find method is the basic substring search operation (it returns the offset of the passed-in substring, or –1 if it is not present), and the string replace method performs global searches and replacements; both act on the subject that they are attached to and called from:

```
>>> S = 'Spam'
>>> S.find('pa') # Find the offset of a substring in S
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ') # Replace occurrences of a string in S with another
'SXYZm'
>>> S
'Spam'
```

```
>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',') # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'spam'
>>> S.upper() # Upper- and lowercase conversions
'SPAM'
>>> S.isalpha() # Content tests: isalpha, isdigit, etc.
True
>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line.rstrip() # Remove whitespace characters on the right side
'aaa,bbb,ccccc,dd'
>>> line.rstrip().split(',') # Combine two operations
['aaa', 'bbb', 'ccccc', 'dd']
```

Notice the last command here—it strips before it splits because Python runs from left to right, making a temporary result along the way. Strings also support an advanced substitution operation known as formatting, available as both an expression (the original) and a string method call (new as of 2.6 and 3.0); the second of these allows you to omit relative argument value numbers as of 2.7 and 3.1:

```
>> '%s, eggs, and %s' % ('spam', 'SPAM!') # Formatting expression (all)
'spam, eggs, and SPAM!'
>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!') # Formatting method (2.6+, 3.0+)
'spam, eggs, and SPAM!'
>>> '{}, eggs, and {}'.format('spam', 'SPAM!') # Numbers optional (2.7+, 3.1+)
'spam, eggs, and SPAM!'
```

Formatting is rich with features, which we’ll postpone discussing until later in this book, and which tend to matter most when you must generate numeric reports:

```
>>> '{:,.2f}'.format(296999.2567) # Separators, decimal digits
'296,999.26'
>>> '%.2f | %+05d' % (3.14159, −42) # Digits, padding, signs
'3.14 | −0042'
```