5. Data Types and Dynamic Typing

Python has a large number of built-in data types, such as Numbers (Integer, Float, Boolean, Complex Number), String, List, Tuple, Set, Dictionary and File. More high-level data types, such as Decimal and Fraction, are supported by external modules.

You can use the built-in function type(*varName*) to check the type of a variable or literal.

5.1 Number Types

Python supports these built-in number types:

1. Integers (type int): e.g., 123, -456. Unlike C/C++/Java, integers are of unlimited size in Python. For example,

```
>>> 123 + 456 - 789
-210
>>> 12345678901234567890123456789012345678901234567890 + 1
12345678901234567890123456789012345678901234567891
>>> 12345678901234567890123456789012345678901234567890 + 1
12345678901234567890123456789012345678901234567891
>>> 2 ** 888    # Raise 2 to the power of 888
.....
>>> len(str(2 ** 888))  # Convert integer to string and get its length
268             # 2 to the power of 888 has 268 digits
>>> type(123)   # Get the type
<class 'int'>
>>> help(int)   # Show the help menu for type int
```

   You can also express integers in hexadecimal with prefix 0x (or 0X); in octal with prefix 0o (or 0O); and in binary with prefix 0b (or 0B). For examples, 0x1abc, 0X1ABC, 0o1776, 0b11000011.

14. Floating-point numbers (type float): e.g., 1.0, -2.3, 3.4e5, -3.4E-5, with a decimal point and an optional exponent (denoted by e or E). floats are 64-bit double precision floating-point numbers. For example,

```
>>> 1.23 * -4e5
-492000.0
>>> type(1.2)      # Get the type
<class 'float'>
>>> import math    # Using the math module
>>> math.pi
3.141592653589793
>>> import random  # Using the random module
>>> random.random()  # Generate a random number in [0, 1)
0.890839384187198
```

24. Booleans (type bool): takes a value of either True or False. Take note of the spelling in initial-capitalized.

```
>>> 8 == 8    # Compare
True
>>> 8 == 9
False
>>> type(True)  # Get type
<class 'bool'>
>>> type (8 == 8)
<class 'bool'>
```

In Python, integer 0, an empty value (such as empty string '', "", empty list [], empty tuple (), empty dictionary {}), and None are treated as False; anything else are treated as True.

```
>>> bool(0)  # Cast int 0 to bool
False
>>> bool(1)  # Cast int 1 to bool
True
>>> bool('') # Cast empty string to bool
False
>>> bool('hello')  # Cast non-empty string to bool
True
>>> bool([])  # Cast empty list to bool
False
>>> bool([1, 2, 3])  # Cast non-empty list to bool
True
```

Booleans can also act as integers in arithmetic operations with 1 for True and 0 for False. For example,

```
>>> True + 3
4
>>> False + 1
1
```

31. Complex Numbers (type complex): e.g., 1+2j, -3-4j. Complex numbers have a real part and an imaginary part denoted with suffix of j (or J). For example,

```
>>> x = 1 + 2j  # Assign variable x to a complex number
>>> x         # Display x
(1+2j)
>>> x.real     # Get the real part
1.0
>>> x.imag     # Get the imaginary part
2.0
>>> type(x)    # Get type
<class 'complex'>
>>> x * (3 + 4j)  # Multiply two complex numbers
(-5+10j)
```

42. Other number types are provided by external modules, such as decimal module for decimal fixed-point numbers, fraction module for rational numbers.

```
# floats are imprecise
>>> 0.1 * 3
0.30000000000000004

# Decimal are precise
>>> import decimal  # Using the decimal module
>>> x = decimal.Decimal('0.1')  # Construct a Decimal object
>>> x * 3   # Multiply  with overloaded * operator
Decimal('0.3')
  >>> type(x)  # Get type
  <class 'decimal.Decimal'>
```

5.2 Dynamic Typing and Assignment Operator

Recall that Python is *dynamic typed* (instead of *static typed*).

Python associates types with objects, instead of variables. That is, a variable does not have a fixed type and can be assigned an object of any type. A variable simply provides a *reference* to an object.

You do not need to declare a variable before using a variable. A variable is created automatically when a value is first assigned, which links the assigned object to the variable.

You can use built-in function type(*var_name*) to get the object type referenced by a variable.

```
>>> x = 1       # Assign an int value to create variable x
>>> x           # Display x
1
>>> type(x)     # Get the type of x
<class 'int'>
>>> x = 1.0     # Re-assign a float to x
>>> x
1.0
>>> type(x)     # Show the type
<class 'float'>
>>> x = 'hello'  # Re-assign a string to x
>>> x
'hello'
>>> type(x)     # Show the type
<class 'str'>
>>> x = '123'   # Re-assign a string (of digits) to x
>>> x
'123'
>>> type(x)     # Show the type
<class 'str'>
```

Type Casting: int(*x*), float(*x*), str(*x*)

You can perform type conversion (or type casting) via built-in functions int(*X*), float(*X*), str(*X*), bool(*X*), etc. For example,

```
>>> x = '123'   # string
>>> type(x)
<class 'str'>
>>> x = int(x)   # Parse str to int, and assign back to x
>>> x
123
>>> type(x)
<class 'int'>
>>> x = float(x)  # Convert x from int to float, and assign back to x
>>> x
123.0
>>> type(x)
<class 'float'>
>>> x = str(x)   # Convert x from float to str, and assign back to x
>>> x
'123.0'
>>> type(x)
<class 'str'>
>>> len(x)      # Get the length of the string
5
>>> x = bool(x)  # Convert x from str to boolean, and assign back to x
```

```
>>> x          # Non-empty string is converted to True
True
>>> type(x)
<class 'bool'>
>>> x = str(x)    # Convert x from bool to str
>>> x
'True'
```

In summary, a variable does not associate with a type. Instead, a type is associated with an object. A variable provides a reference to an object (of a certain type).

Check Instance's Type: isinstance(*instance*, *type*)

You can also use the built-in function isinstance(*instance*, *type*) to check if the instance belong to the type. For example,

```
>>> isinstance(123, int)
True
>>> isinstance('a', int)
False
>>> isinstance('a', str)
True
```

The Assignment Operator (=)

In Python, you do not need to *declare* variables before using the variables. The initial assignment creates a variable and links the assigned value to the variable. For example,

```
>>> x = 8              # Create a variable x by assigning a value
>>> x = 'Hello'        # Re-assign a value (of a different type) to x
>>> y                  # Cannot access undefined (unassigned) variable
NameError: name 'y' is not defined
```

Pair-wise Assignment and Chain Assignment

For example,

```
>>> a = 1  # Ordinary assignment
>>> a
1
>>> b, c, d = 123, 4.5, 'Hello'  # Pair-wise assignment of 3 variables and values
>>> b
123
>>> c
4.5
>>> d
'Hello'
>>> e = f = g = 123  # Chain assignment
>>> e
123
>>> f
123
>>> g
123
```

Assignment operator is *right-associative*, i.e., a = b = 123 is interpreted as (a = (b = 123)).

del Operator

You can use del operator to delete a variable. For example,

```
>>> x = 8    # Create variable x via assignment
>>> x
8
>>> del x    # Delete variable x
>>> x
NameError: name 'x' is not defined
```

5.3 Number Operations

Arithmetic Operators (+, -, *, /, //, **, %)

Python supports these arithmetic operators:

Operator	Mode	Usage	Description	Example
+	Binary Unary	*x + y* *+x*	Addition Positive	
-	Binary Unary	*x - y* *-x*	Subtraction Negate	
*****	Binary	*x * y*	Multiplication	
/	Binary	*x / y*	Float Division (Returns a float)	1 / 2 ⇒ 0.5 -1 / 2 ⇒ -0.5
//	Binary	*x // y*	Integer Division (Returns the floor integer)	1 // 2 ⇒ 0 -1 // 2 ⇒ -1 8.9 // 2.5 ⇒ 3.0 -8.9 // 2.5 ⇒ -4.0 (floor!) -8.9 // -2.5 ⇒ 3.0
******	Binary	*x ** y*	Exponentiation	2 ** 5 ⇒ 32 1.2 ** 3.4 ⇒ 1.858729691979481
%	Binary	*x % y*	Modulus (Remainder)	9 % 2 ⇒ 1 -9 % 2 ⇒ 1 9 % -2 ⇒ -1 -9 % -2 ⇒ -1 9.9 % 2.1 ⇒ 1.5 -9.9 % 2.1 ⇒ 0.6000000000000001

Compound Assignment Operators (+=, -=, *=, /=, //=, **=, %=)

Each of the arithmetic operators has a corresponding *shorthand assignment* counterpart, i.e., +=, -=, *=, /=, //=, **= and %=. For example i += 1 is the same as i = i + 1.

Increment/Decrement (++, --)?

Python does not support increment (++) and decrement (--) operators (as in C/C++/Java). You need to use i = i + 1 or i += 1 for increment.

Python accepts ++i ⇒ +(+i) ⇒ i, and --i. Don't get trap into this. But Python flags a syntax error for i++ and i--.

Mixed-Type Operations

For mixed-type operations, e.g., 1 + 2.3 (int + float), the value of the "smaller" type is first promoted to the "bigger" type. It then performs the operation in the "bigger" type and returns the result in the "bigger" type. In Python, int is "smaller" than float, which is "smaller" than complex.

Relational (Comparison) Operators (==, !=, <, <=, >, >=, in, not in, is, is not)

Python supports these relational (comparison) operators that return a bool value of either True or False.

Operator	Mode	Usage	Description	Example
== **!=** **<** **<=** **>** **>=**	Binary	*x* == *y* *x* != *y* *x* < *y* *x* <= *y* *x* > *y* *x* >= *y*	Comparison Return bool of either True or False	
in **not in**	Binary	*x* in *seq* *x* not in *seq*	Check if *x* is contained in the sequence *y* Return bool of either True or False	lst = [1, 2, 3] x = 1 x in lst ⇒ False
is **is not**	Binary	*x* is *y* *x* is not *y*	Check if *x* and *y* are referencing the same object Return bool of either True or False	

Logical Operators (and, or, not)

Python supports these logical (boolean) operators, that operate on boolean values.

Operator	Mode	Usage	Description	Example
and	Binary	*x* and *y*	Logical AND	
or	Binary	*x* or *y*	Logical OR	
not	Unary	not *x*	Logical NOT	

Notes:

- Python's logical operators are typed out in word, unlike C/C++/Java which uses symbols &&, || and !.
- Python does not have an exclusive-or (xor) boolean operator.

Built-in Functions

Python provides many built-in functions for numbers, including:

- Mathematical functions: round(), pow(), abs(), etc.
- type() to get the type.
- Type conversion functions: int(), float(), str(), bool(), etc.
- Base radix conversion functions: hex(), bin(), oct().

For examples,

```
# Test built-in function round()
>>> x = 1.23456
>>> type(x)
<type 'float'>

# Python 3
>>> round(x)    # Round to the nearest integer
1
>>> type(round(x))
<class 'int'>

# Python 2
>>> round(x)
1.0
>>> type(round(x))
<type 'float'>

>>> round(x, 1)  # Round to 1 decimal place
1.2
>>> round(x, 2)  # Round to 2 decimal places
1.23
>>> round(x, 8)  # No change - not for formatting
1.23456

# Test other built-in functions
>>> pow(2, 5)
32
>>> abs(-4.1)
4.1

# Test base radix conversion
>>> hex(1234)
'0x4d2'
>>> bin(254)
'0b11111110'
>>> oct(1234)
'0o2322'
>>> 0xABCD  # Shown in decimal by default
43981

# List built-in functions
>>> dir(__built-ins__)
['type', 'round', 'abs', 'int', 'float', 'str', 'bool', 'hex', 'bin', 'oct',.....]

# Show number of built-in functions
```

```
>>> len(dir(__built-ins__))  # Python 3
151
>>> len(dir(__built-ins__))  # Python 2
144

# Show documentation of __built-ins__ module
>>> help(__built-ins__)
.....
```

Bitwise Operators (Advanced)

Python supports these bitwise operators:

Operator	Mode	Usage	Description	Example x=0b10000001 y=0b10001111
&	binary	*x* & *y*	bitwise AND	x & y ⇒ 0b10000001
|	binary	*x* ! *y*	bitwise OR	x | y ⇒ 0b10001111
~	Unary	~*x*	bitwise NOT (or negate)	~x ⇒ -0b10000010
^	binary	*x* ^ *y*	bitwise XOR	x ^ y ⇒ 0b00001110
<<	binary	*x* << *count*	bitwise Left-Shift (padded with zeros)	x << 2 ⇒ 0b1000000100
>>	binary	*x* >> *count*	bitwise Right-Shift (padded with zeros)	x >> 2 ⇒ 0b100000

5.4 String

In Python, strings can be delimited by a pair of single-quotes ('...') or double-quotes ("..."). Python also supports multi-line strings via triple-single-quotes ('''...''') or triple-double-quotes ("""...""").
To place a single-quote (') inside a single-quoted string, you need to use escape sequence \'. Similarly, to place a double-quote (") inside a double-quoted string, use \". There is no need for escape sequence to place a single-quote inside a double-quoted string; or a double-quote inside a single-quoted string.
A triple-single-quoted or triple-double-quoted string can *span multiple lines*. There is no need for escape sequence to place a single/double quote inside a triple-quoted string. Triple-quoted strings are useful for multi-line documentation, HTML and other codes.

Python 3 uses Unicode character set to support internationalization (i18n).

```
>>> s1 = 'apple'
>>> s1
'apple'
>>> s2 = "orange"
>>> s2
'orange'
>>> s3 = "'orange'"  # Escape sequence not required
>>> s3
"'orange'"
>>> s3 ="\"orange\""  # Escape sequence needed
>>> s3
'"orange"'
```

```
# A triple-single/double-quoted string can span multiple lines
>>> s4 = """testing
12345"""
>>> s4
'testing\n12345'
```

Escape Sequences for Characters (\code)

Like C/C++/Java, you need to use escape sequences (a back-slash + a code) for:

- Special non-printable characters, such as tab (\t), newline (\n), carriage return (\r)
- Resolve ambiguity, such as \" (for " inside double-quoted string), \' (for ' inside single-quoted string), \\ (for \).
- \xhh for character in hex value and \ooo for octal value
- \uxxxx for 4-hex-digit (16-bit) Unicode character and \Uxxxxxxxx for 8-hex-digit (32-bit) Unicode character.

Raw Strings (r'...' or r"...")

You can prefix a string by r to disable the interpretation of escape sequences (i.e., \code), i.e., r'\n' is '\'+'n' (two characters) instead of newline (one character). Raw strings are used extensively in regex (to be discussed in module re section).

Strings are Immutable

Strings are *immutable*, i.e., their contents cannot be modified. String functions such as upper(), replace() returns a new string object instead of modifying the string under operation.

Built-in Functions and Operators for Strings

You can operate on strings using:

- built-in functions such as len();
- operators such as in (contains), + (concatenation), * (repetition), indexing [*i*] and [-*i*], and slicing **[*m*:*n*:*step*]**.

Note: These functions and operators are applicable to all sequence data types including string, list, and tuple (to be discussed later).

Function / Operator	Usage	Description	Examples s = 'Hello'
len()	len(*str*)	Length	len(s) ⇒ 5
in	*substr* in *str*	Contain? Return bool of either True or False	'ell' in s ⇒ True 'he' in s ⇒ False
+ **+=**	*str* + *str1* *str* += *str1*	Concatenation	s + '!' ⇒ 'Hello!'
***** ***=**	*str* * *count* *str* *= *count*	Repetition	s * 2 ⇒ 'HelloHello'
[*i*] **[-*i*]**	*str*[*i*] *str*[-*i*]	Indexing to get a character. The front index begins at 0; back index begins at -1 (=len(*str*)-1).	s[1] ⇒ 'e' s[-4] ⇒ 'e'
[*m*:*n*:*step*] **[*m*:*n*]** **[*m*:]**	*str*[*m*:*n*:*step*] *str*[*m*:*n*] *str*[*m*:]	Slicing to get a substring. From index *m* (included)	s[1:3] ⇒ 'el' s[1:-2] ⇒ 'el' s[3:] ⇒ 'lo'

Function / Operator	Usage	Description	Examples s = 'Hello'
[:*n*] **[:]**	*str*[:*n*] *str*[:]	to *n* (excluded) with *step* size. The defaults are: *m*=0, *n*=-1, *step*=1.	s[:-2] ⇒ 'Hel' s[:] ⇒ 'Hello' s[0:5:2] ⇒ 'Hlo'

For examples,

```
>>> s = "Hello, world"  # Assign a string literal to the variable s
>>> type(s)          # Get data type of s
<class 'str'>
>>> len(s)     # Length
12
>>> 'ello' in s  # The in operator
True

# Indexing
>>> s[0]     # Get character at index 0; index begins at 0
'H'
>>> s[1]
'e'
>>> s[-1]    # Get Last character, same as s[len(s) - 1]
'd'
>>> s[-2]    # 2nd last character
'l'

# Slicing
>>> s[1:3]    # Substring from index 1 (included) to 3 (excluded)
'el'
>>> s[1:-1]
'ello, worl'
>>> s[:4]    # Same as s[0:4], from the beginning
'Hell'
>>> s[4:]    # Same as s[4:-1], till the end
'o, world'
>>> s[:]     # Entire string; same as s[0:len(s)]
'Hello, world'

# Concatenation (+) and Repetition (*)
>>> s = s + " again"  # Concatenate two strings
>>> s
'Hello, world again'
>>> s * 3       # Repeat 3 times
'Hello, world againHello, world againHello, world again'

# str can only concatenate with str, not with int and other types
>>> s = 'hello'
>>> print('The length of \'' + s + '\' is ' + len(s))  # len() is int
TypeError: can only concatenate str (not "int") to str
>>> print('The length of \'' + s + '\' is ' + str(len(s)))
The length of 'hello' is 5

# String is immutable
```

```
>>> s[0] = 'a'
TypeError: 'str' object does not support item assignment
```

Character Type?

Python does not have a dedicated character data type. A character is simply a string of length 1. You can use the indexing operator to extract individual character from a string, as shown in the above example; or process individual character using for-in loop (to be discussed later).

The built-in functions ord() and chr() operate on character, e.g.,

```
# ord(c) returns the integer ordinal (Unicode) of a one-character string
>>> ord('A')
65
>>> ord('水')
27700

# chr(i) returns a one-character string with Unicode ordinal i; 0 <= i <= 0x10ffff.
>>> chr(65)
'A'
>>> chr(27700)
'水'
```

Unicode vs ASCII

In Python 3, strings are defaulted to be Unicode. ASCII strings are represented as byte strings, prefixed with b, e.g., b'ABC'.

In Python 2, strings are defaulted to be ASCII strings (byte strings). Unicode strings are prefixed with u.

You should always use Unicode for internationalization (i18n)!

String-Specific Member Functions

Python supports strings via a built-in class called str (We will describe class in the Object-Oriented Programming chapter). The str class provides many member functions. Since string is immutable, most of these functions return a new string. The commonly-used member functions are as follows, supposing that *s* is a strobject:

- *s*.strip(), *s*.rstrip(), *s*.lstrip(): strip the leading and trailing whitespaces, the right (trailing) whitespaces; and the left (leading) whitespaces, respectively.
- *s*.upper(), *s*.lower(): Return a uppercase/lowercase counterpart, respectively.
- *s*.isupper(), *s*.islower(): Check if the string is uppercase/lowercase, respectively.
- *s*.find(*key_str*):
- *s*.index(*key_str*):
- *s*.startswith(*key_str*):
- *s*.endswith(*key_str*):
- *s*.split(*delimiter_str*), *delimiter_str*.join(*strings_list*):

```
>>> dir(str)     # List all attributes of the class str
[..., 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> s = 'Hello, world'
>>> type(s)
<class 'str'>
```

```
>>> dir(s)       # List all attributes of the object s
.......
>>> help(s.find)  # Show the documentation of member function find
.......
>>> s.find('ll')  # Find the beginning index of the substring
2
>>> s.find('app')  # find() returns -1 if not found
-1

>>> s.index('ll')  # index() is the same as find(), but raise ValueError if not found
2
>>> s.index('app')
ValueError: substring not found

>>> s.startswith('Hell')
True
>>> s.endswith('world')
True
>>> s.replace('ll', 'xxx')
'Hexxxo, world'
>>> s.isupper()
False
>>> s.upper()
'HELLO, WORLD'

>>> s.split(', ')   # Split into a list with the given delimiter
['Hello', 'world']
>>> ', '.join(['hello', 'world', '123'])  # Join all strings in the list using the delimiter
'hello, world, 123'

>>> s = '  testing 12345   '
>>> s.strip()      # Strip leading and trailing whitespaces
'testing 12345'
>>> s.rstrip()     # Strip trailing (right) whitespaces
'  testing 12345'
>>> s.lstrip()     # Strip leading (left) whitespaces
'testing 12345   '

# List all the whitespace characters - in module string, attribute whitespace
>>> import string
>>> string.whitespace   # All whitespace characters
' \t\n\r\x0b\x0c'
>>> string.digits      # All digit characters
'0123456789'
>>> string.hexdigits    # All hexadecimal digit characters
'0123456789abcdefABCDEF'
```

String Formatting 1 (New Style): Using str.format() function

There are a few ways to produce a formatted string for output. Python 3 introduces a new style in the str's format() member function with {} as place-holders (called format fields). For examples,

```
# Replace format fields {} by arguments in format() in the SAME order
>>> '|{}|{}|more|'.format('Hello', 'world')
'|Hello|world|more|'

# You can use 'positional' index in the form of {0}, {1}, ...
>>> '|{0}|{1}|more|'.format('Hello', 'world')
'|Hello|world|more|'
>>> '|{1}|{0}|more|'.format('Hello', 'world')
'|world|Hello|more|'

# You can use 'keyword' inside {}
>>> '|{greeting}|{name}|'.format(greeting='Hello', name='Peter')
'|Hello|Peter|'

# Mixing 'positional' and 'keyword'
>>> '|{0}|{name}|more|'.format('Hello', name='Peter')
'|Hello|Peter|more|'
>>> '|{}|{name}|more|'.format('Hello', name='Peter')
'|Hello|Peter|more|'

# You can specify field-width with :n,
#  alignment (< for left-align, > for right-align, ^ for center-algin) and
#  padding (or fill) character.
>>> '|{1:8}|{0:7}|'.format('Hello', 'Peter')   # Set field-width
'|Peter   |Hello  |'     # Default left-aligned
>>> '|{1:8}|{0:>7}|{2:-<10}|'.format('Hello', 'Peter', 'again')  # Set alignment and padding
'|Peter   |  Hello|again-----|'    # > (right align), < (left align), - (fill char)
>>> '|{greeting:8}|{name:7}|'.format(name='Peter', greeting='Hi')
'|Hi      |Peter  |'

# Format int using ':d' or ':nd'
# Format float using ':f' or ':n.mf'
>>> '|{0:.3f}|{1:6.2f}|{2:4d}|'.format(1.2, 3.456, 78)
'|1.200|  3.46|  78|'
# With keywords
>>> '|{a:.3f}|{b:6.2f}|{c:4d}|'.format(a=1.2, b=3.456, c=78)
'|1.200|  3.46|  78|'
```

When you pass lists, tuples, or dictionaries (to be discussed later) as arguments into the format() function, you can reference the sequence's elements in the format fields with [index]. For examples,

```
# list and tuple
>>> tup = ('a', 11, 22.22)
>>> tup = ('a', 11, 11.11)
>>> lst = ['b', 22, 22.22]
>>> '|{0[2]}|{0[1]}|{0[0]}|'.format(tup)  # {0} matches tup, indexed via []
'|11.11|11|a|'
>>> '|{0[2]}|{0[1]}|{0[0]}|{1[2]}|{1[1]}|{1[0]}|'.format(tup, lst)  # {0} matches tup, {1} matches lst
'|11.11|11|a|22.22|22|b|'

# dictionary
```

```
>>> dict = {'c': 33, 'cc': 33.33}
>>> '|{0[cc]}|{0[c]}|'.format(dict)
'|33.33|33|'
>>> '|{cc}|{c}|'.format(**dict)  # As keywords via **
'|33.33|33|'
```

String Formatting 2: Using *str*.rjust(*n*), *str*.ljust(*n*), *str*.center(*n*), *str*.zfill(*n*)

You can also use str's member functions like *str*.rjust(*n*) (where *n* is the field-width), *str*.ljust(*n*), *str*.center(*n*), *str*.zfill(*n*) to format a string. For example,

```
# Setting field width and alignment
>>> '123'.rjust(5)
'  123'
>>> '123'.ljust(5)
'123  '
>>> '123'.center(5)
' 123 '
>>> '123'.zfill(5)  # Pad (Fill) with leading zeros
'00123'

# Floats
>>> '1.2'.rjust(5)
'  1.2'
>>> '-1.2'.zfill(6)
'-001.2'
```

String Formatting 3 (Old Style): Using % operator

The old style (in Python 2) is to use the % operator, with C-like printf() format specifiers. For examples,

```
# %s for str
# %ns for str with field-width of n (default right-align)
# %-ns for left-align
>>> '|%s|%8s|%-8s|more|' % ('Hello', 'world', 'again')
'|Hello|   world|again   |more|'

# %d for int
# %nd for int with field-width of n
# %f for float
# %n.mf for float with field-with of n and m decimal digits
>>> '|%d|%4d|%6.2f|' % (11, 222, 33.333)
'|11| 222| 33.33|'
```

Avoid using old style for formatting.

Conversion between String and Number: int(), float() and str()

You can use built-in functions int() and float() to parse a "numeric" string to an integer or a float; and str() to convert a number to a string. For example,

```
# Convert string to int
>>> s = '12345'
>>> s
'12345'
>>> type(s)
<class 'str'>
>>> i = int(s)
>>> i
12345
>>> type(i)
<class 'int'>

# Convert string to float
>>> s = '55.66'
>>> s
'55.66'
>>> f = float(s)
>>> f
55.66
>>> type(f)
<class 'float'>
>>> int(s)
ValueError: invalid literal for int() with base 10: '55.66'

# Convert number to string
>>> i = 123
>>> s = str(i)
>>> s
'123'
>>> type(s)
<class 'str'>
'123'
```

Concatenate a String and a Number?

You CANNOT concatenate a string and a number (which results in TypeError). Instead, you need to use the str() function to convert the number to a string. For example,

```
>>> 'Hello' + 123
TypeError: cannot concatenate 'str' and 'int' objects
>>> 'Hello' + str(123)
'Hello123'
```

5.5 The None Value

Python provides a special value called None (take note of the spelling in initial-capitalized), which can be used to initialize an object (to be discussed in OOP later). For example,

```
>>> x = None
>>> type(x)   # Get type
<class 'NoneType'>
>>> print(x)
None

# Use 'is' and 'is not' to check for 'None' value.
>>> print(x is None)
True
>>> print(x is not None)
False
```