Data structure and algorithm in Python

Python Primer

Xiaoping Zhang

School of Mathematics and Statistics, Wuhan University

Table of contents

1. Python Overview
2. Objects in Python
3. Expressions, Operators, and Percedence
4. Control Flow
5. Functions
6. Simple Input and Output
7. Exception Handling
8. Iterators and Generators
9. Additional Python Conveniences
10. Scopes and Namespaces
11. Modules and the Import Statement

Python Overview

Python Overview

The Python Interpreter

The Python Interpreter

```
print('Welcome to the GPA calculator.')
print('Please enter all your letter grades, one per line.')
print('Enter a blank line to designate the end.')
# map from letter grade to point value
points = {'A+':4.0,  'A':4.0, 'A-':3.67,
          'B+':3.33, 'B':3.0, 'B-':2.67,
          'C+':2.33, 'C':2.0, 'C' :1.67,
          'D+':1.33, 'D':1.0, 'F' :0.0}
num_courses = 0
total_points = 0
done = False
while not done:
  grade = input()    # read line from user
  if grade == '':    # line was entered
    done = True
  elif grade not in points:  # unrecognized grade entered
    print("Unknown grade '{0}' being ignored".format(grade))
  else:
    num_courses += 1
    total_points += points[grade]
if num_courses > 0:   # division by zero
  print('Your GPA is {0:.3}'.format(total_points / num_courses))
```

Objects in Python

Python is an object-oriented language and classes form the basis for all data types.

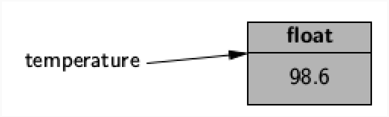
- key aspects of Python's object model
- built-in classes
    - `int` class for integers
    - `float` class for floating-point values
    - `str` class for character strings

Objects in Python

Idetifiers, Objects, and the Assignment Statement

Assignment Statement

```
temperature = 98.6
```



Identifiers

- Case-sensitive
- Composed of almost any combination of letters, numerals, and underscore characters
- Cannot begin with a numeral
- 33 specially reserved words that cannot be used as identifiers

Identifiers

False	as	continue	else	from	in	not
return	yield	None	assert	def	except	global
is	or	try	True	break	del	finally
if	lambda	pass	while	and	class	elif
for	import	nonlocal	raise	with		

Identifiers

Python is a dynamically typed language, as there is no advance declaration associating an identifier with a particular data type.

- An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type.
- Although an identifier has no declared type, the object to which it refers has a definite type.

Identifiers

A programmer can establish an alias by assigning a second identifier to an existing object. Once an alias has been established, either name can be used to access the underlying object.

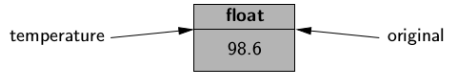


Figure 1.2: Identifiers temperature and original are aliases for the same object.

Identifiers

if one of the names is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias.

```
temperature += 5.0
```

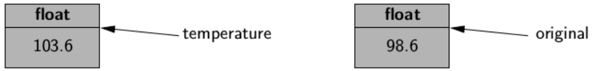


Figure 1.3: The temperature identifier has been assigned to a new value, while original continues to refer to the previously existing value.

Objects in Python

Creating and Using Objects

Instantiation

The process of creating a new instance of a class is known as instantiation. In general, the syntax for instantiating an object is to invoke the constructor of a class.

Example

Given a class named `Widget`, we could create an instance using

- `w = Widget()`, if the constructor does not require any parameters
- `w = Widget(a, b, c)`, if the constructor does require parameters

Instantiation

Many of Python's built-in classes support what is known as a literal form for designating new instances.

Example

The command `temperature = 98.6` results in the creation of a new instance of the `float` class; the term 98.6 in that expression is a literal form.

Instantiation

From a programmer's perspective, yet another way to indirectly create a new instance of a class is to call a function that creates and returns such an instance.

Example

Python has a built-in function named `sorted` that takes a sequence of comparable elements as a parameter and returns a new instance of the list class containing those elements in sorted order.

Calling Methods

Python's classes may define one or more methods (also known as member functions), which are invoked on a specific instance of a class using the dot (".") operator.

Example

Python's list class has a method named sort that can be invoked with a syntax such as `data.sort()`. This particular method rearranges the contents of the list so that they are sorted.

Calling Methods

The expression to the left of the dot identifies the object upon which the method is invoked. Often, this will be an identifier (e.g., data), but we can use the dot operator to invoke a method upon the immediate result of some other operation.

Example

If `response` identifies a string instance, the syntax

```
response.lower().startswith('y')
```

first evaluates the method call `response.lower()`, which itself returns a new string instance, and then the method `startswith('y')` is called on the intermediate string.

Calling Methods

When using a method of a class, it is important to understand its behavior.

- Some methods return information about the state of an object, but do not change that state. These are known as accessors.
- Other methods, such as the sort method of the list class, do change the state of an object. These methods are known as mutators or update methods.

Objects in Python

Python's Built-in Classes

Python’s Built-in Classes

A class is immutable if each object of that class has a fixed value upon instantiation that cannot subsequently be changed.

Example

The float class is immutable. Once an instance has been created, its value cannot be changed, although an indetifier referencing that object can be reassigned to a different value.

Python's Built-in Classes

Class	Description	Immutable?
`bool`	Boolean value	Yes
`int`	integer	Yes
`float`	floating-point number	Yes
`list`	mutable sequence of objects	
`tuple`	immutable sequence of objects	Yes
`str`	character string	Yes
`set`	unordered set of distinct objects	
`dict`	associate mapping (dictionary)	
`frozenset`	immutable form of set class	Yes

Objects in Python

The bool class

The bool class

The **bool** class is used to manipulate logical (Boolean) values, and the only two instances of that class are expressed as the literals ***True*** and ***False***.

The bool class

- The default constructor, **bool( )**, returns False, but there is no reason to use that syntax rather than the more direct literal form.
- Python allows the creation of a Boolean value from a nonboolean type using the syntax **bool(foo)** for value foo. The interpretation depends upon the type of the parameter.
  - Numbers evaluate to False if zero, and True if nonzero.
  - Sequences and other container types, such as strings and lists, evaluate to False if empty and True if nonempty.

Objects in Python

The int class

The int class

The **int** and **float** classes are the primary numeric types in Python.

The **int** class is designed to represent integer values with arbitrary magnitude.

The int class

- The interger constructor, **int( )**, returns value 0 by default.
- But this constructor can be used to construct an integer value based upon an existing value of another type.
  - If f represents a floating-point value, the syntax int(f) produces the truncated value of f.
  - `int(3.14)-> 3, int(3.99)-> 3, int(-3.9)-> -3`
  - If s represents a string, then int(s) produces the integral value that string represents.
    `int('137')-> 137, int('7f', 16)-> 127`

Objects in Python

The float class

The float class

The float class is the sole floating-point type in Python, using a fixed-precision representation. float type.

The float class

- The interger constructor, **float( )**, returns value 0.0.
- When given a parameter, the con- structor attempts to return the equivalent floating-point value.
  - `float(2)-> 2.0`
  - If the parameter to the constructor is a string, as with float('3.14'), it attempts to parse that string as a floating-point value, raising a ValueError as an exception.

Objects in Python

Sequence Types: The list, tuple, and str Classes

Sequence Types: The list, tuple, and str Classes

The list, tuple, and str classes are sequence types in Python, representing a collection of values in which the order is significant.

- The **list** class is the most general, representing a sequence of arbitrary objects (akin to an "array" in other languages).
- The **tuple** class is an immutable version of the list class, benefiting from a streamlined internal representation.
- The **str** class is specially designed for representing an immutable sequence of text characters.

List

A list instance stores a sequence of objects.

- A list is a referential structure, as it technically stores a sequence of references to its elements.
- Lists are array-based sequences and are zero-indexed, thus a list of length n has elements indexed from 0 to n-1 inclusive.
- Python uses the characters [ ] as delimiters for a list literal, with [ ] itself being an empty list.

```
['red', 'green', 'blue']
```

List

The **list( )** constructor produces an empty list by default. However, the constructor will accept any parameter that is of an iterable type.

Example

`list('hello')` produces a list of individual characters,

`['h', 'e', 'l', 'l', 'o']`

Tuple

The tuple class provides an immutable version of a sequence, and therefore its instances have an internal representation that may be more streamlined than that of a list.

- While Python uses the [ ] characters to delimit a list, parentheses delimit a tuple, with ( ) being an empty tuple.
- There is one important subtlety. To express a tuple of length one as a literal, a comma must be placed after the element, but within the parentheses.

```
(17, )   # one element tuple
(17)     # a simple numeric
```

Str

Python’ s str class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international character set. Strings have a more compact internal representation than the referential lists and tuples.

Set and frozenset

Python’ s set class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements.

Python uses curly braces `{` and `}` as delimiters for a set, for example, as `{17}` or `{'red', 'green', 'blue'}`.

Dict

Python’ s dict class represents a dictionary, or mapping, from a set of distinct keys to associated values.

A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form `{ }` produces an empty dictionary. A nonempty dictionary is expressed using a comma-separated series of **key:value** pairs.

`{’ga’: ’Irish’, ’de’: ’German}`.

Expressions, Operators, and Percedence

Expressions, Operators, and Percedence

Logical Operators

Logical Operators

`not`	unary negation
`and`	conditional and
`or`	conditional or

Expressions, Operators, and Percedence

Equality Operators

Equality Operators

`is`	same identity
`is not`	different identity
`==`	equivalent
`!=`	not equivalent

In most programming situations,

- the equivalence tests `==` and `!=` are the appropriate operators;
- use of `is` and `is not` should be reserved for situations in which it is necessary to detect true aliasing.

Expressions, Operators, and Percedence

Comparison Operators

Comparison Operators

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Expressions, Operators, and Percedence

Arithmetic Operators

Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	true division
//	integer division
%	the modulo operator

Arithmetic Operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	true division
`//`	integer division
`%`	the modulo operator

Consider $27 \div 4$

- math: $27 \div 4 = 6\frac{3}{4} = 6.75$
- python:
  - `27 / 4 == 6.75`
  - `27 // 4 == 6` (mathematical floor of the quotient)
  - `27 % 4 == 3`

Remark: C, C++, and Java do not support the // operator

Expressions, Operators, and Percedence

Bitwise Operator

Bitwise Operator

`~`	bitwise complement
`&`	bitwise and
`|`	bitwise or
`^`	bitwise exclusive-or
`<<`	shift bits left, filling in with zeros
`>>`	shift bits right, filling in with sign bit

Expressions, Operators, and Percedence

Sequence Operator

Sequence Operator

Each of Python's built-in sequence types (str, tuple and list) support the following operator syntaxes:

`s[j]`	element at index j
`s[start:stop]`	slice including indices [start, stop)
`s[start:stop:step]`	slice including indices start, start+step, start+2*step, ..., up to but not equal to stop
`s+t`	concatenation of sequences
`k*s`	shorthand for s+s++...(k times)
`val in s`	containment check
`val not in s`	non-containment check

Expressions, Operators, and Percedence

Operator for Sets and Dictionaries

Operator for Sets and Dictionaries

Sets and frozensets support the following operators:

`key in s`	containment check
`key not in s`	non-containment check
`s1 == s2`	s1 is equivalent to s2
`s1 != s2`	s1 is not equivalent to s2
`s1 <= s2`	s1 is subset of s2
`s1 < s2`	s1 is proper subset of s2
`s1 >= s2`	s1 is superset of s2
`s1 > s2`	s1 is proper superset of s2
`s1 | s2`	the union of s1 and s2
`s1 & s2`	the intersection of s1 and s2
`s1 - s2`	the set of elements in s1 but not s2
`s1 ^ s2`	the set of elements in precisely one of s1 or s2

Operator for Sets and Dictionaries

Dictionaries support the following operators:

`d[key]`	value associated with given key
`d[key] = value`	set (or reset) the value associated with given key
`del d[key]`	remove key and its associated value from dictionary
`key in d`	containment check
`key not in d`	non-containment check
`d1 == d2`	d1 is equivalent to d2
`d1 != d2`	d1 is not equivalent to d2

Expressions, Operators, and Percedence

Extended Assignment Operators

Extended Assignment Operators

Python supports an extended assignment operator for most binary operators, such as `count += 5`, which is a shorthand for `count = count + 5`.

```
a = [1, 2, 3]
b = a
b += [4, 5]
b = b + [6, 7]
print(a)
```

Expressions, Operators, and Percedence

Compound Expressions and Operator Precedence

Compound Expressions and Operator Precedence

- Allows a chained assignment, such as `x = y = 0`
- Allows the chaining of comparison operators, such as `1 <= x+y <= 10`

Compound Expressions and Operator Precedence

	Type	Symbols
1	member access	`expr.member`
2	function/method calls	`expr(...)`
	container subscripts/slides	`expr[...]`
3	exponentiation	`**`
4	unary operators	`+expr, -expr, ~expr`
5	multiplication, division	`*, /, //, %`
6	addition, subtraction	`+, -`
7	bitwise shifting	`<<, >>`
8	bitwise-and	`&`
9	bitwise-xor	`^`
10	bitwise-or	`|`

Compound Expressions and Operator Precedence

	Type	Symbols
11	comparisons	`is`, `is not`, `==`, `!=`, `<`, `<=`, `>`, `>=`
	containment	`in`, `not in`
12	logical-not	`not expr`
13	logical-and	`and`
14	logical-or	`or`
15	conditional	`val1 if condition else val2`
16	assignments	`=`, `+=`, `-=`, `*=`, `etc`

Control Flow

Control Flow

Conditionals

Conditionals

```
if first_condition:
  first_body
elif second_condition:
  second_body
elif third_condition:
  third_body
else:
  fourth_body
```

Conditionals

Example

A robot controller might have the following logic:

```
if door_is_closed:
  open_door()
advance()
```

Conditionals

Example

We may nest one control structure within another, relying on indentation to make clear the extent of the various bodies.

```
if door_is_closed:
  if door_is_locked:
    unlock_door()
  open_door()
advance()
```

Conditionals

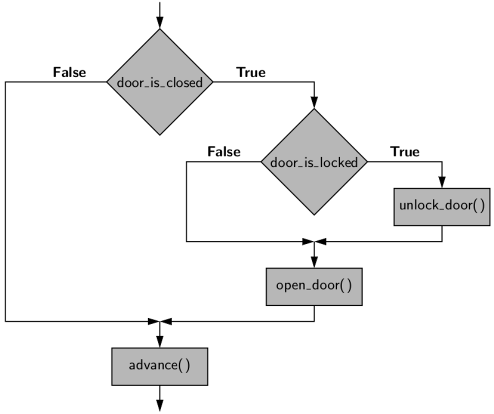


Figure 1.6: A flowchart describing the logic of nested conditional statements.

Control Flow

Loops

Loops

Python offers two distinct looping constructs.

- A **while** loop allows general repetition based upon the repeated testing of a Boolean condition.
- A **for** loop provides convenient iteration of values from a defined series (such as characters of a string, elements of a list, or numbers within a given range).

While Loops

- Syntax

```
while condition:
  body
```

- Example

```
j = 0
while j < len(data) and data[j] != 'X'
  j += 1
```

For Loops

The **for-loop** syntax can be used on any type of iterable structure, such as a list, tuple str, set, dict, or file.

- Syntax

```
for element in iterable:
  body
```

- Example 1 (task of computing the sum of a list of numbers)

```
total = 0
for val in data:
  total += val
```

- Example 2 (task of finding the maximum value of a list of elements)

```
biggest = data[0]
for val in data:
  if val > biggest:
    biggest = val
```

Index-Based For Loops

In some cases, we prefer to loop over all possible indices of the list. For this purpose, Python provides a built-in class named **range** that generates integer sequences.

```
big index = 0
for j in range(len(data)):
  if data[j] > data[big_index]:
    big_index = j
```

Break and Continue Statements

Python supports a break statement that immediately terminate a while or for loop when executed within its body.

Example

```
# determines whether a target value occurs
# in a data set
found = False;
for item in data:
  if item == target:
    found = True
    break
```

Break and Continue Statements

Python also supports a continue statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

Example

```
for x in range(7):
  if (x == 3 or x==6):
    continue
  print(x)
```

Functions

Functions

A distinction between **functions** and **methods**:

- **function**: a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as `sorted(data)`.
- **method**: a member function that is invoked upon a specific object using an object-oriented message passing syntax, such as `data.sort()`.

Functions

```python
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

Functions

Return Statement

Return Statement

A return statement is used within the body of a function to indicate that the function should immediately cease execution, and that an expressed value should be returned to the caller. If a return statement is executed without an explicit argument, the **None** value is automatically returned.

Example

```
def contains(data, target):
  for item in target:
    if item == target:     # found a match
      return True
  return False
```

Functions

Information Passing

Information Passing

In the context of a function signature, the identifiers used to describe the expected parameters are known as **formal parameters**, and the objects sent by the caller when invoking the function are the **actual parameters**. Parameter passing in Python follows the semantics of the standard **assignment statement**.

Information Passing

Example

```
prizes = count(grades, 'A')
```

Just before the function body is executed, the actual parameters, grades and A , are implicitly assigned to the formal parameters, data and target, as follows:

```
data = grades
target = 'A'
```

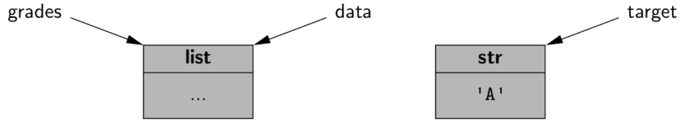


Figure 1.7: A portrayal of parameter passing in Python, for the function call count(grades, 'A'). Identifiers data and target are formal parameters defined within the local scope of the count function.

Functions

Default Parameters Values

Default Parameters Values

Python provides means for functions to support more than one possible calling signature. Such a function is said to be polymorphic (which is Greek for “many forms”). Most notably, functions can declare one or more default values for paameters, thereby allowing the caller to invoke a function with varying numbers of actual parameters.

Default Parameters Values

Example

If a function is declared with signature

```
def foo(a, b=15, c=27):
```

there are three parameters, the last two of which offer default values.

Three calling syntax

- `foo(4, 12, 8)`
- `foo(4)`
- `foo(8, 20)`

Default Parameters Values

Example

Illegal function definition:

```
def bar(a, b=15, c)
```

where `b` has a default value, yet not the subsequent `c`.

If a default parameter value is present for one parameter, it must be present for all further parameters.

Default Parameters Values

Example (An interesting polymorphic function `range`)

Three calling syntax:

- `range(n)`
- `range(start, stop)`
- `range(start, stop, step)`

Default Parameters Values

Example (An interesting polymorphic function `range`)

Three calling syntax:

- `range(n)`
- `range(start, stop)`
- `range(start, stop, step)`

This combination of forms seems to violate the rules for default parameters. In particular, when a single parameter is sent, as in `range(n)`, it serves as the stop value (which is the second parameter); the value of `start` is effectively 0 in that case.

Default Parameters Values

Example (An interesting polymorphic function `range`)

Three calling syntax:

- `range(n)`
- `range(start, stop)`
- `range(start, stop, step)`

This combination of forms seems to violate the rules for default parameters. In particular, when a single parameter is sent, as in `range(n)`, it serves as the stop value (which is the second parameter); the value of `start` is effectively 0 in that case.

Default Parameters Values

This effect can be achieved as follows:

```python
def range(start, stop=None, step=1):
  if stop is None:
    stop = start
    start = 0
  ...
```

Functions

Keyword Parameters

Keyword Parameters

The traditional mechanism for matching the actual parameters sent by a caller, to the formal parameters declared by the function signature is based on the concept of **positional arguments**.

Example (positional arguments)

With signature `foo(a=10, b=20, c=30)`, parameters sent by the caller are matched, in the given order, to the formal parameters. An invocation of `foo(5)` indicates that `a=5`, while `b` and `c` are assigned their default values.

Keyword Parameters

Python supports an alternate mechanism for sending a parameter to a function known as a **keyword argument**. A keyword argument is specified by explicitly assigning an actual parameter to a formal parameter by name.

Example

keyword argument With the above definition of function foo, a call `foo(c=5)` will invoke the function with parameters `a=10, b=20, c=5`.

Keyword Parameters

```
>>> def f(a, b, c=1):
...     return a*b+c
...
>>> print(f(1, 2))
3
>>> print(f(1, 2, 3))
5
>>> print(f(1, a=2, 3))
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>> print(f(1, b=2, c=3))
5
>>> print(f(a=5, b=2, c=2))
12
>>> print(f(c=5, a, b))
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>> print(f(b=2, a=2))
5
```

Functions

Python's Built-in Functions

Python's Built-in Functions

Calling Syntax	Description
`abs(x)`	Return the absolute value of a number.
`all(iterable)`	Return `True` if `bool(e)` is True for each element `e`.
`any(iterable)`	Return `True` if `bool(e)` is True for at least one element `e`.
`chr(integer)`	Return a one-character string with the given Unicode code point.
`divmod(x, y)`	Return `(x // y, x % y)` as tuple, if `x` and `y` are integers.
`hash(obj)`	Return an integer hash value for the object.
`id(obj)`	Return the unique integer serving as an "identity" for the object.

Python's Built-in Functions

Calling Syntax	Description
`input(prompt)`	Return a string from standard input; the prompt is optional.
`isinstance(obj, cls)`	Determine if `obj` is an instance of the class (or a subclass).
`iter(iterable)`	Return a new iterator object for the parameter.
`len(iterable)`	Return the number of elements in the given iteration.
`map(f, iter1, iter2, ...)`	Return an iterator yielding the result of function calls `f(e1, e2, ...)` for respective elements `e1` ∈ `iter1`, `e2` ∈ `iter2`, ...

Python’s Built-in Functions

Calling Syntax	Description
`max(iterable)`	Return the largest element of the given iteration.
`max(a, b, c, ...)`	Return the largest of the arguments.
`min(iterable)`	Return the smallest element of the given iteration.
`min(a, b, c, ...)`	Return the smallest of the arguments.
`next(iterator)`	Return the next element reported by the iterator
`open(filename, mode)`	Open a file with the given name and access mode
`ord(char)`	Return the Unicode code point of the given character

Python’s Built-in Functions

Calling Syntax	Description
`pow(x, y)`	Return the value x^y (as an integer if x and y are integers); equivalent to x**y
`pow(x, y, z)`	Return $x^y \mod z$ as an integer
`print(obj1, obj2, ...)`	Print the arguments, with seperating spaces and trailing newline
`range(stop)`	Construct an iteration of values 0,1,...,stop-1
`range(start, stop)`	Construct an iteration of values start, start+1, ..., stop-1
`range(start, stop, step)`	Construct an iteration of values start, start+step, start+2*step, ...

Python's Built-in Functions

Calling Syntax	Description
`reversed(sequence)`	Return an iteration of the sequence in reverse
`round(x)`	Return the nearest in value
`round(x, k)`	Return the value rounded to the nearest 10^{-k}
`sorted(iterable)`	Return a list containing elements of the iterable in sorted order
`sum(iterable)`	Return the sum of the elements in the iterable (must be numeric)
`type(obj)`	Return the class to which the instance obj belongs

Simple Input and Output

Simple Input and Output

Console Input and Output

Console Input and Output: print function

The built-in function, `print`, is used to generate standard output to the console.

In its simplest form, it prints an arbitrary sequence of arguments, separated by spaces, and followed by a trailing newline character.

Console Input and Output: print function

By default,

- inserts a separating space into the output between each pair of arguments. The separator can be customized by providing a desired separating string as a keyword parameter, `sep`.

Console Input and Output: print function

- a trailing newline is output after the final argument. An alternative trailing string can be designated using a keyword parameter, `end`.

Console Input and Output: print function

- sends its output to the standard console. However, output can be directed to a file by indicating an output file stream using `file` as a keyword parameter.

Console Input and Output: input function

The built in function `input` displays a prompt, if given as an optional parameter, and then waits until the user enters some sequence of characters followed by the return key.

- The formal return value of the function is the string of characters that were entered strictly before the return key (i.e., no newline character exists in the returned string).
- When reading a numeric value from the user, a programmer must use the input function to get the string of characters, and then use the int or float syntax to construct the numeric value that character string represents.
- Because input returns a string as its result, use of that function can be combined with the existing functionality of the string class.

Console Input and Output: input function

A sample program

```
age = int(input('Enter your age in years: '))
max_heart_rate = 206.9 - (0.67 * age)
target = 0.65 * max_heart_rate
print('Your target fat-burning heat rate is',
target)
```

Simple Input and Output

Files

Files

Files are typically accessed in Python beginning with a call to a built-in function, named `open`, that returns a proxy for interactions with the underlying file.

Example

`fp = open('sample.txt')` attempts to open a file named `sample.txt`, returning a proxy that allows read-only access to the text file.

Files

The `open` function accepts an optional second parameter that determines the access mode.

access mode	meaning
`r`	reading
`w`	writing to the file (causing any existing file with that name to be overwritten)
`a`	appending to the end of an existing file
`rb`	
`wb`	
`ab`	

Files

Calling Syntax	Description
`fp.read()`	Return the (remaining) contents of a readable file as a string.
`fp.read(k)`	Return the next *k* bytes of a readable file as a string.
`fp.readline()`	Return (remainder of) the current line of a readable file as a string.
`fp.readlines()`	Return all (remaining) lines of a readable file as a list of strings.
`for line in fp:`	Iterate all (remaining) lines of a readable file.
`fp.seek(k)`	Change the current position to be at the k-th byte of the file.

Files

Calling Syntax	Description
`fp.tell()`	Return the current position, measured as byte-offset from the start.
`fp.write(string)`	Write given string at current position of the writable file.
`fp.writelines(seq)`	Write each of the strings of the given sequence at the current position of the writable file. This command does not insert any newlines, beyond those that are embedded in the strings.
`print(...,file=fp)`	Redirect output of print function to the file.

Exception Handling

Exception Handling

Class	Description
Exception	A base class for most error types
NameError	Raised if nonexistent identifier used
AttributeError	Raised by syntax `obj.foo`, if obj has no member named foo
TypeError	Raised when wrong type of parameter is sent to a function
EOFError	Raised if “end of file” reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)

Exception Handling

Class	Description
StopIteration	Raised by `next(iterator)` if no element
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
ValueError	Raised when parameter has invalid value (e.g., `sqrt(-5)`)
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Exception Handling

Raising an Exception

Raising an Exception

An exception is thrown by executing the raise statement, with an appropriate instance of an exception class as an argument that designates the problem.

Example

If a function for computing a square root is sent a negative value as a parameter, it can raise an exception with the command:

```
  raise ValueError('x cannot be negative')
```

This syntax raises a newly created instance of the ValueError class, with the error message serving as a parameter to the constructor. If this exception is not caught within the body of the function, the execution of the function immediately ceases and the exception is propagated to the calling context (and possibly beyond).

Raising an Exception

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x cannot be negative')
  # do the real work here
```

Raising an Exception

An implementation with rigorous error-checking might be written as follows:

```
def sum(values):
  if not isinstance(values, collections.Iterable
  ):
    raise TypeError('parameter must be an
    iterable type)
  total = 0
  for v in values:
    if not isinstance(v, (int, float)):
      raise TypeError('elements must be numeric'
      )
    total += v
  return total
```

Raising an Exception

A far more direct and clear implementation of this function can be written as follows:

```
def sum(values):
  total = 0
  for v in values:
    total += v
  return total
```

Exception Handling

Catching an Exception

Catching an Exception

Look before you leap

```
if y != 0:
  ratio = x / y
else:
  ... do something else ...
```

Catching an Exception

It is easier to ask for forgiveness than it is to get permission

```
try:
  ratio = x / y
except ZeroDivisionError:
  ... do something else ...
```

Catching an Exception

```
try:
  fp = open('sample.txt')
except IOError as e:
  print('Unable to open the file:', e)
```

Catching an Exception

```
age = -1
while age <= 0:
  try:
    age = int(input('Enter your age in years:'))
    if age <= 0:
      print('Your age must be positive')
  except(ValueError, EOFError):
    print('Invalid response')
```

Catching an Exception

If we preferred to have the while loop continue without printing the Invalid response message, we could have written the exception-clause as

```
age = -1
while age <= 0:
  try:
    age = int(input('Enter your age in years:'))
    if age <= 0:
      print('Your age must be positive')
  except(ValueError, EOFError):
    pass
```

Catching an Exception

```
age = -1
while age <= 0:
  try:
    age = int(input('Enter your age in years:'))
    if age <= 0:
      print('Your age must be positive')
  except ValueError:
    print('That's an invalid age specification')
  except EOFError:
    print('There was an unexpected error reading
     input')
    raise
```

Catching an Exception

```
try:
  You do your operations here;
  ...
except:
  If there is any exception, then execute this
  block.
  ...
else:
  If there is no exception then execute this
  block.
```

Catching an Exception

```
try:
  You do your operations here;
  ...
  Due to any exception, this may be skipped.
finally:
  This would always be executed.
  ...
```

Iterators and Generators

Iterators and Generators

Iterators

Iterators and Generators

for-loop sytax

```
for element in iterable:
  ...
```

- Basic container types, such as list, tuple, and set, qualify as iterable types.
- Furthermore, a string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines.
- User-defined types may also support iteration.

Iterators and Generators

In Python, the mechanism for iteration is based upon the following conventions:

- An iterator is an object that manages an iteration through a series of values.

  If variable, `i`, identifies an iterator object, then each call to the built-in function, `next(i)`, produces a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.
- An iterable is an object, `obj`, that produces an iterator via the syntax `iter(obj)`.

By these definitions, an instance of a list is an iterable, but not itself an iterator.

Iterators and Generators

Generators

Generators

The most convenient technique for creating iterators in Python is through the use of **generators**.

A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series.

Generators

Example (Determine all factors of a positive integer)

A traditional function might produce and return a list containing all factors, implemented as:

```
def factors(n):
  results = []
  for k in range(1, n+1):
    if n % k == 0
      results.append(k)
  return results
```

Generators

In contrast, an implementation of a generator for computing those factors could be implemented as follows:

```
def factors(n):
  for k in range(1, n+1):
    if n % k == 0
      yield k
```

Generators

```
def factors(n):
  k = 1
  while k * k < n:
    if n % k == 0:
      yield k
      yield n // k
  if k*k == n:
      yield k
```

Generators

```
def fibonacci():
  a = 0
  b = 1
  while True:
    yield a
    future = a + b
    a = b
    b = future
```

Additional Python Conveniences

Additional Python Conveniences

Conditional Expressions

Conditional Expressions

A **conditional expression** syntax that can replace a simple control structure.

```
expr1 if condition else expr2
```

This compound expression evaluates to `expr1` if the condition is true, and otherwise evaluates to `expr2`. It is equivalent to the syntax `condition ? expr1 : expr2` in C/C++.

Conditional Expressions

Example (find the absolute value of a variable)

traditional control structure

```
def abs(n):
  if n >= 0:
    return n
  else:
    return -n
result = abs(-4)
```

conditional expression syntax

```
def abs(n):
  return n if n >= 0 else -n
result = abs(-4)
```

Additional Python Conveniences

Comprehension Syntax

Comprehension Syntax

General form of list comprehension

```
[expression for value in iterable if condition]
```

traditional control structure

```
result = []
for value in iterable:
  if condition:
    result.append(expression)
```

Comprehension Syntax

Example (Generate a list of squares of the numbers from 1 to n)

traditional control structure

```
squres = []
for k in range(1, n+1):
  squares.append(k*k)
```

list comprehension

```
squares = [k*k for k in range(1, n+1)]
```

Comprehension Syntax

Example (Produce a list of factors for an integer *n*)

list comprehension

```
factors = [k for k in range(1, n+1) if n % k ==
0]
```

Comprehension Syntax

Python supports similar comprehension syntaxes that respectively produce a set, generator, or dictionary. We compare those syntaxes using our example for producing the squares of numbers.

```
# list comprehension
[k*k for k in range(1, n+1)]

# set comprehension
{k*k for k in range(1, n+1)}

# generator comprehension
(k*k for k in range(1, n+1))

# dictionary comprehension
{k: k*k for k in range(1, n+1)}
```

Additional Python Conveniences

Packing and Unpacking of Sequences

Packing and Unpacking of Sequences

Python provides two additional conveniences involving the treatment of tuples and other sequence types.

- **Automatic packing** of a tuple
  - The assignment

    ```
    data = 2, 4, 6, 8
    ```

    results in identifier, `data`, being assigned to the tuple `(2, 4, 6, 8)`.
  - If the body of a function executes the command,

    ```
    return x, y
    ```

    it will be formally returning a single object that is the tuple `(x, y)`.

Packing and Unpacking of Sequences

- **Automatically unpack** a sequence
  - The assignment

    ```
    a, b, c, d = range(7, 11)
    ```

    has the effect of assigning `a = 7, b = 8, c = 9, d = 10`.
  - The built-in function, `divmod(a, b)`, returns the pair of values `(a // b, a % b)`. It is possible to write

    ```
    quotient, remainder = divmod(a, b)
    ```

    to separately identify the two entries of the returned tuple.
  -

    ```
    for x, y in [(7, 2), (5, 8), (6, 4)]:
        ...

    for key, value in mapping.items():
        ...
    ```

Additional Python Conveniences

Simultaneous Assignments

Simultaneous Assignments

The combination of automatic packing and unpacking forms a technique known as **simultaneous assignment**.

- Explicitly assign a series of values to a series of identifiers:

```
x, y, z = 6, 2, 5
```

- A convinient way for swapping the values associated with two variables:

```
j, k = k, j
```

Simultaneous Assignments

The use of simultaneous assignments can greatly simplify the presentation of code.

Example (The generator pruducing Fibonacci series)

With simultaneous assignements, the generator can be implemented more directly as follows:

```
def fibonacci():
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a+b
```

Scopes and Namespaces

Scopes and Namespaces

Whenever an identifier is assigned to a value, that definition is made with a specific ***scope***.

- Top-level assignments are typically made in what is known as***global scope***.
- Assignments made within the body of a function typically have scope that is ***local*** to that function call.

Scopes and Namespaces

Each distinct scope in Python is represented using an abstraction known as a ***namespace***. A namespace manages all identifiers that are currently defined in given scope.

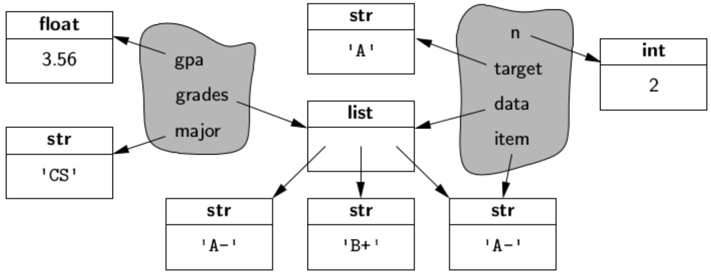


Figure 1.8: A portrayal of the two namespaces associated with a user's call count(grades, 'A'), as defined in Section 1.5. The left namespace is the caller's and the right namespace represents the local scope of the function.

Scopes and Namespaces

Python implements a namespace with its own dictionary that maps each identifying string to its associated value. Python provides several ways to examine a given namespace.

- `dir()`: reports the names of the identifiers in a given namespace (i.e., the keys of the dictionary).
- `vars()`: returns the full dictionary.

Scopes and Namespaces

First-class objects

First-class objects

In the terminology of programming languages, ***first-class object*** are instances of a type that can be ared ṭo an identifier, passed as a parameter, or returned by a function.

First-class objects

```
>>> def foo(text):
...     return len(text)
...
>>> foo("hello")
5
>>> id(foo)
140068385921632
>>> type(foo)
<class 'function'>
>>> foo
<function foo at 0x7f6436631a60>
```

First-class objects

```
>>> bar = foo
>>> bar
<function foo at 0x7f6436631a60>
>>> bar("hello")
5
>>> a = foo
>>> b = foo
>>> c = foo
>>> a is b is c
True
```

First-class objects

```
>>> funcs = [foo, str, len]
>>> funcs
[<function foo at 0x7f6436631a60>, <class 'str'
>, <built-in function len>]
>>> funcs[0]("Python")
6
```

First-class objects

```
>>> def show(func):
...     size = func("Python")
...     print("Length of string is : %d" % size)
...
>>> show(foo)
Length of string is : 6
```

First-class objects

```
>>> def nick():
...     return foo
...
>>> nick
<function nick at 0x7f6436631b70>
>>> a = nick()
>>> a
<function foo at 0x7f6436631a60>
>>> a("Python")
6
```

Modules and the Import Statement

Modules and the Import Statement

Depending on the version of Python, there are approximately 130–150 definitions that were deemed significant enough to be included in that built-in namespace.

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', ..., 'str'
, 'sum', 'super', 'tuple', 'type', 'vars', 'zip'
]
```

Modules and the Import Statement

Beyond the built-in definitions, the standard Python distribution includes per- haps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as modules, that can be imported from within a program.

Modules and the Import Statement

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '
__package__', '__spec__', 'acos', 'acosh', 'asin
', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', '
copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
 'erfc', 'exp', 'expm1', 'fabs', 'factorial', '
floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
 'isnan', 'ldexp', 'lgamma', 'log', 'log10', '
log1p', 'log2', 'modf', 'nan', 'pi', 'pow', '
radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
```

Modules and the Import Statement

Python’s import statement loads definitions from a module into the current namespace.

- Syntax 1:

```
>>> from math import pi, sqrt
>>> angle = pi/2
>>> v = sqrt(2)
```

Modules and the Import Statement

- Syntax 2: If there are many definitions from the same module to be imported, an asterisk may be used as a wild card

```
>>> from math import *
```

Modules and the Import Statement

- Syntax 3:

```
>>> import math
```

Modules and the Import Statement

Creating a New Module

Creating a New Module

To create a new module, one simply has to put the relevant definitions in a file named with a `.py` suffix. Those definitions can be imported from any other `.py` file within the same project directory.

utils.py

```
def count(data, target):
    n = 0
    for item in data:
        if item == target:
            n += 1
    return n
```

```
>>> from utils import count
>>> count([1,2,2,2,3], 2)
2
```

Modules and the Import Statement

Existing Modules

Existing Modules

Module Name	Description
array	Compact array storage for primitive types.
collections	Additional data structures and abstract base classes involving collections of objects.
copy	General functions for making copies of objects.
heapq	Heap-based priority queue functions.
math	Common mathematical constants and functions.
os	Support for interactions with the operating system.
random	Random number generation.
re	Support for processing regular expressions.
sys	Additional level of interaction with the Python interpreter.
time	Support for measuring time, or delaying a program.