Programming Lab

Autumn Semester

Course code: PC503

I can toast!
Let me rephrase that
I can do ANYTHING you tell me to...
... provided you speak my language

Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 7

Different Data Types

- The keyword ***def introduces a function definition***. It must be followed by the function name and the parenthesized list of formal parameters.

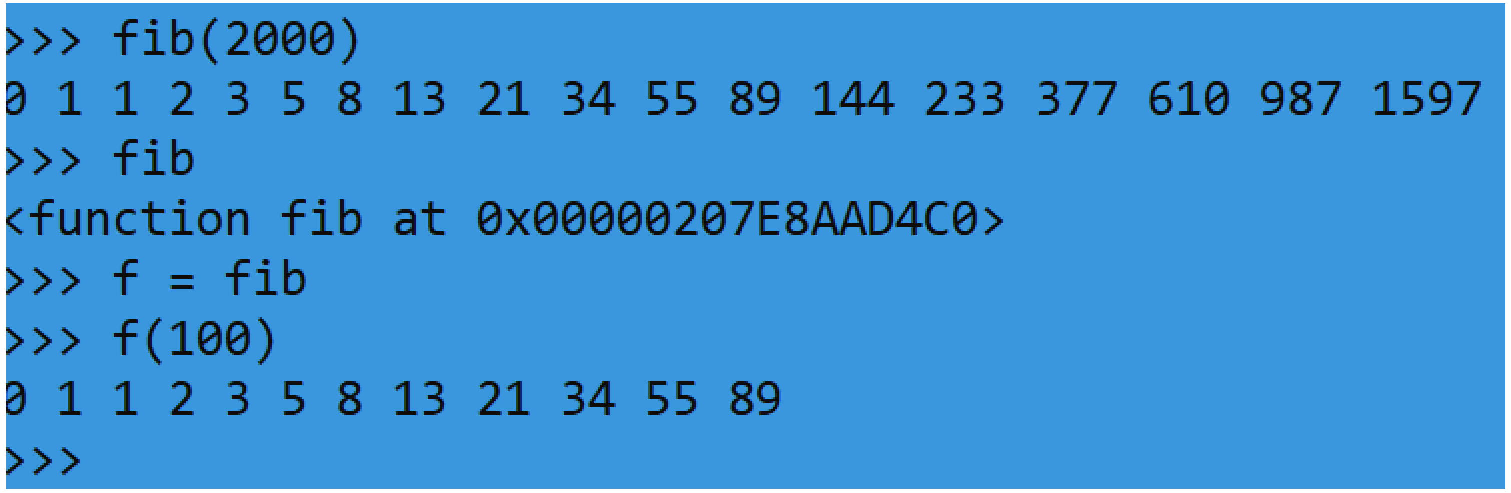
- The statements that form the body of the function start at the ***next line and must be indented.***

- The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string ***or docstring.***

- There are tools that use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; *it’s good practice to include docstrings in code that you write, so make a habit of it.*

- The *execution* of a function introduces a new symbol table used for the local variables of the function.

- More precisely, ***all variable assignments in a function store the value in the local symbol table***; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names.

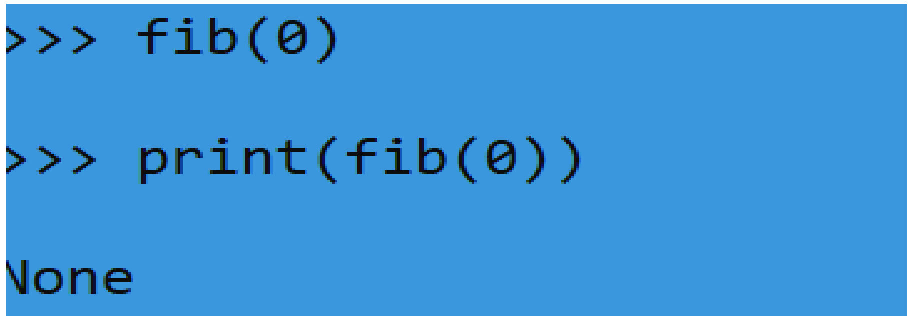


```
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>> fib
<function fib at 0x00000207E8AAD4C0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>>
```

- Coming from other languages, you might object that *fib* is not a function but a procedure since it doesn’t return a value.

- In fact, even functions without a return statement do return a value, albeit a rather boring one. *This value is called None* (it’s a built-in name).

- Writing the value None is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to use ***print():***

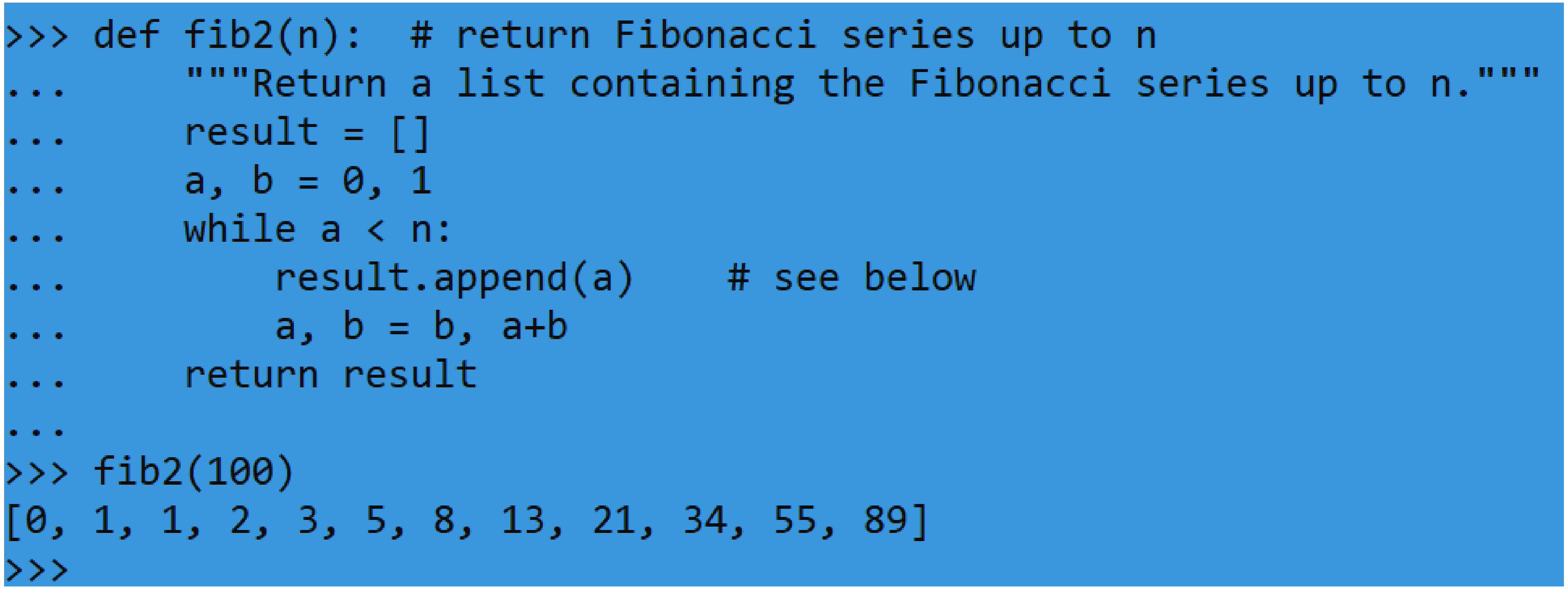


```
>>> fib(0)

>>> print(fib(0))

None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:



```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>
```

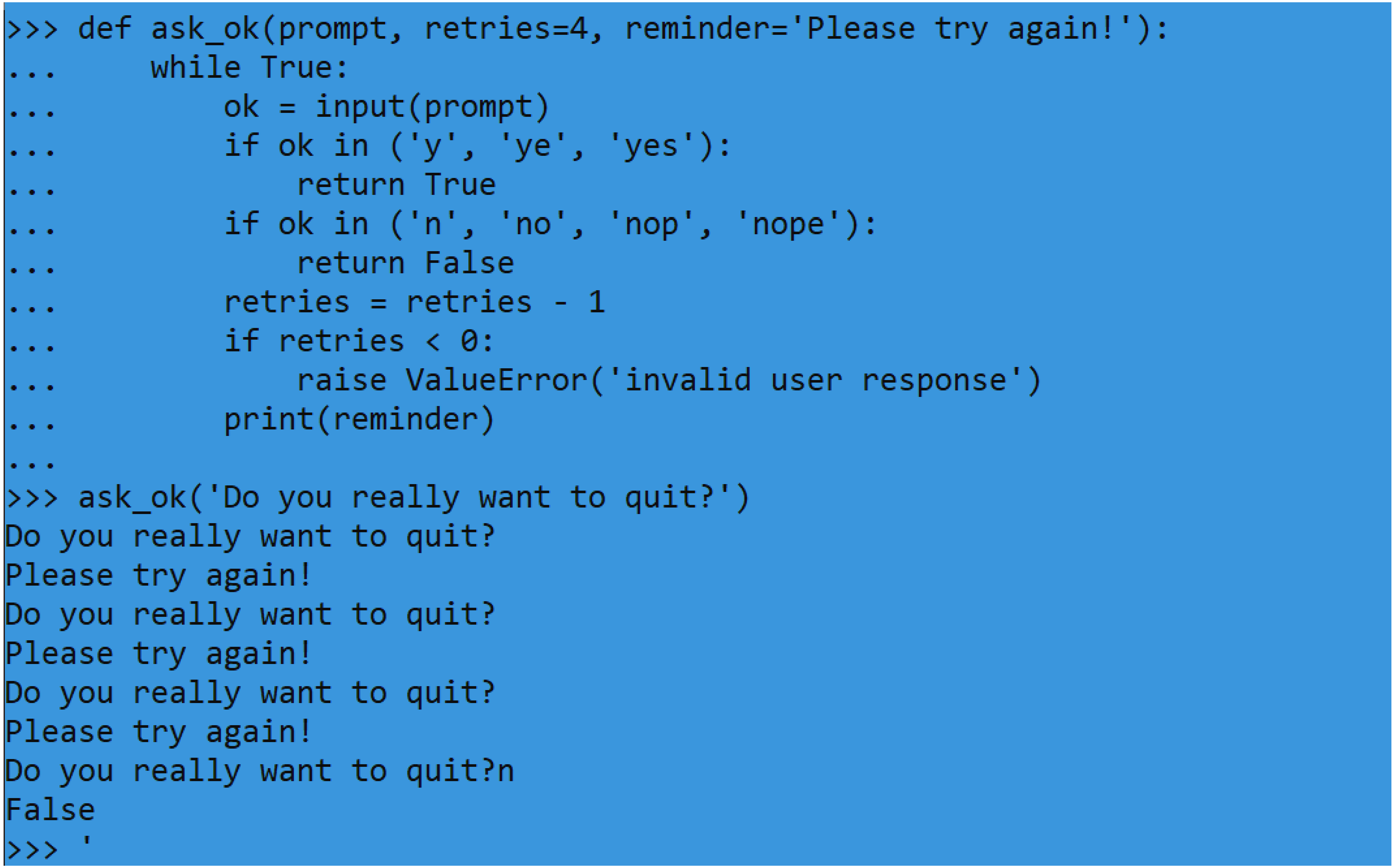
More on Defining Functions

It is also possible to define functions with a variable number of arguments.

There are three forms, which can be combined.

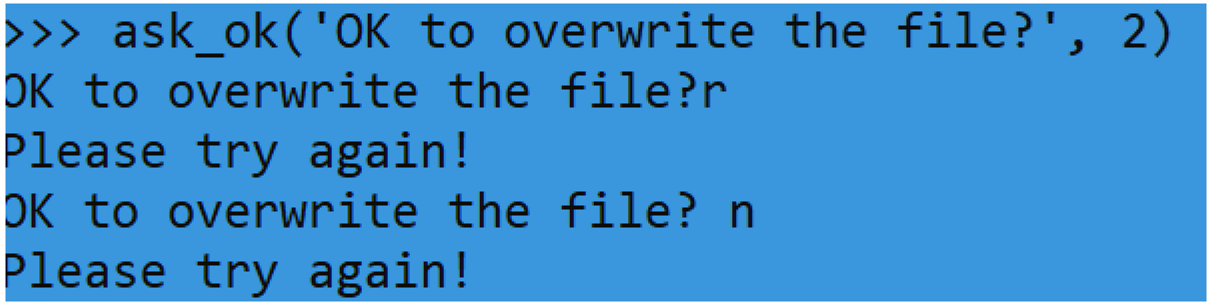
1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

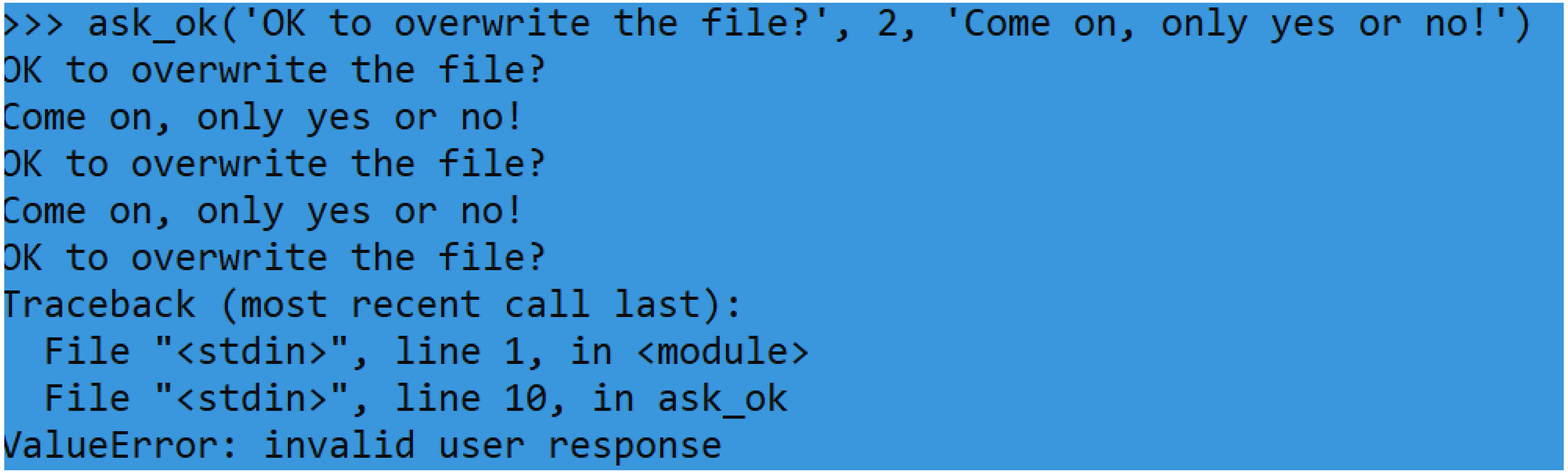


```
>>> def ask_ok(prompt, retries=4, reminder='Please try again!'):
...     while True:
...         ok = input(prompt)
...         if ok in ('y', 'ye', 'yes'):
...             return True
...         if ok in ('n', 'no', 'nop', 'nope'):
...             return False
...         retries = retries - 1
...         if retries < 0:
...             raise ValueError('invalid user response')
...         print(reminder)
...
>>> ask_ok('Do you really want to quit?')
Do you really want to quit?
Please try again!
Do you really want to quit?
Please try again!
Do you really want to quit?
Please try again!
Do you really want to quit?n
False
>>> '
```

More on Defining Functions



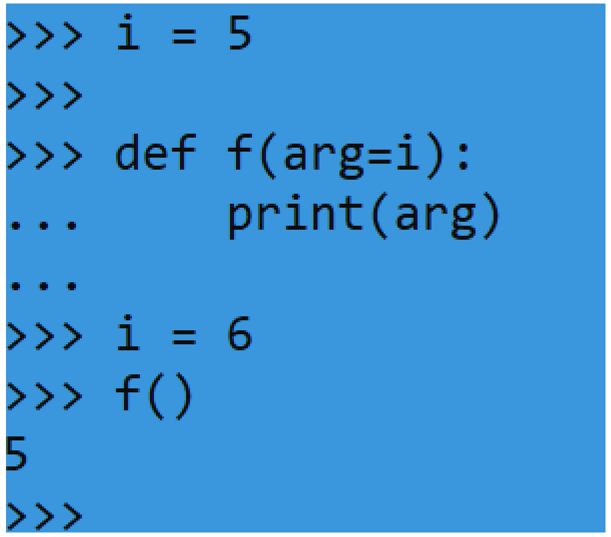
```
>>> ask_ok('OK to overwrite the file?', 2)
OK to overwrite the file?r
Please try again!
OK to overwrite the file? n
Please try again!
```



```
>>> ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
OK to overwrite the file?
Come on, only yes or no!
OK to overwrite the file?
Come on, only yes or no!
OK to overwrite the file?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in ask_ok
ValueError: invalid user response
```

More on Defining Functions

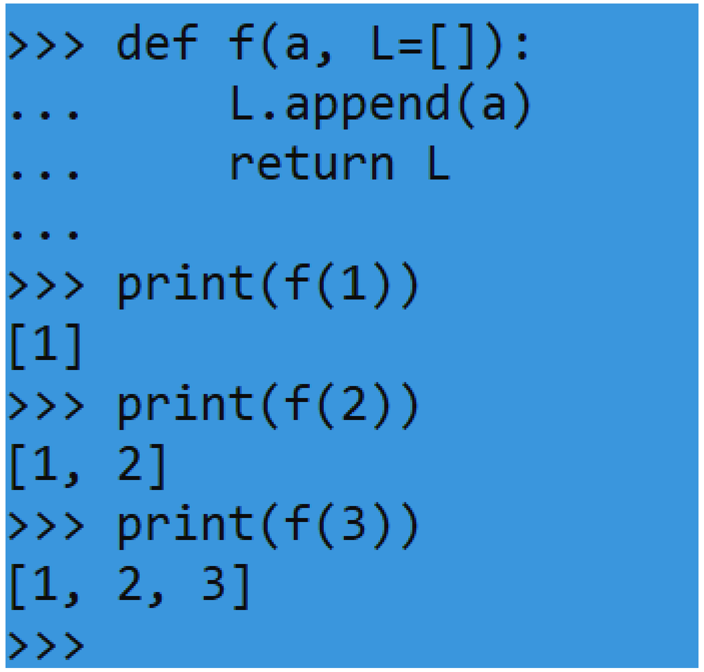
The default values are evaluated at the point of function definition in the defining scope, so that



```
>>> i = 5
>>>
>>> def f(arg=i):
...     print(arg)
...
>>> i = 6
>>> f()
5
>>>
```

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

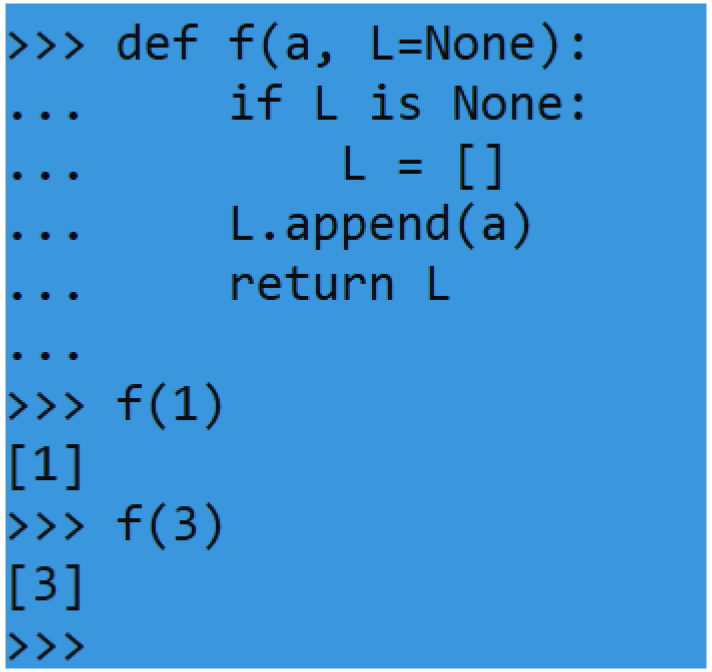
For example, the following function accumulates the arguments passed to it on subsequent calls:



```
>>> def f(a, L=[]):
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
>>>
```

More on Defining Functions

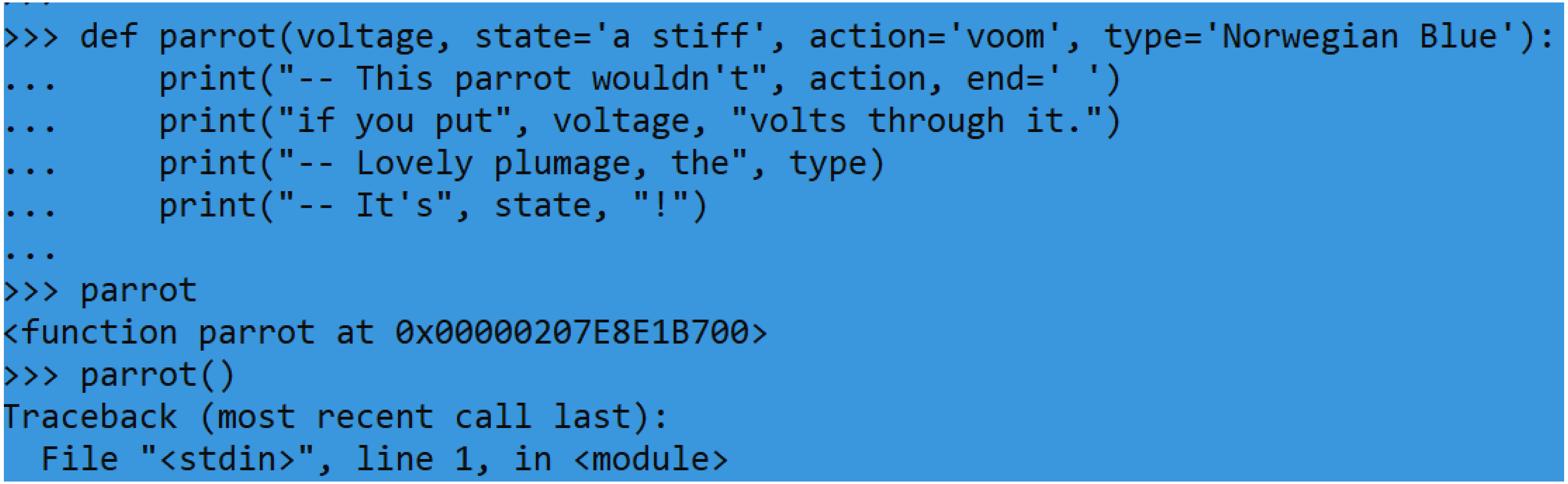
If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:



```
>>> def f(a, L=None):
...     if L is None:
...         L = []
...     L.append(a)
...     return L
...
>>> f(1)
[1]
>>> f(3)
[3]
>>>
```

Keyword Arguments

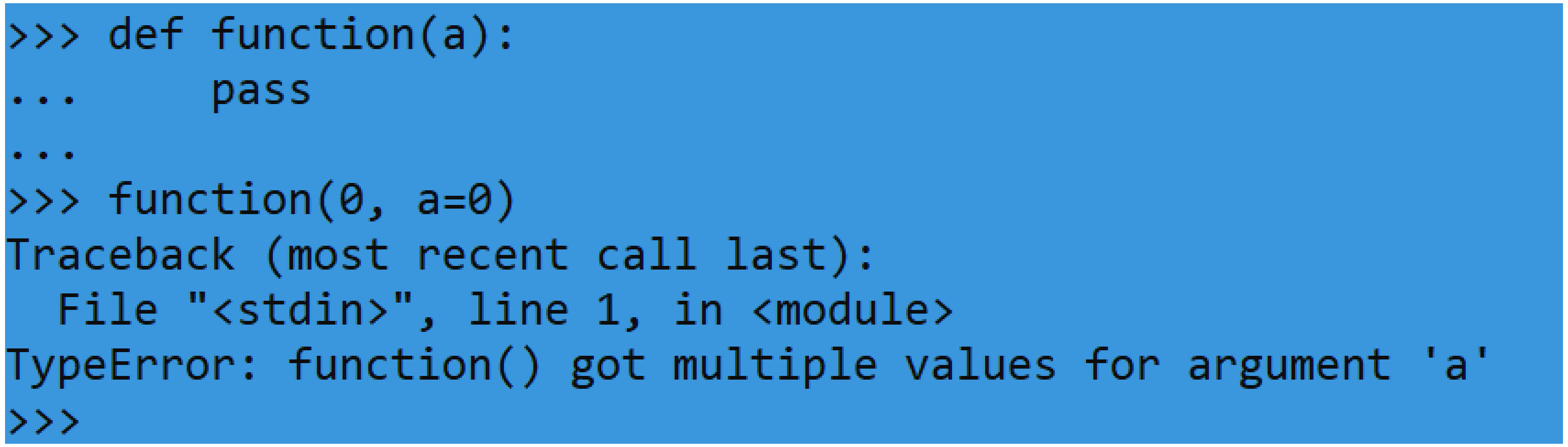
Functions can also be called using **keyword** arguments of the form *kwarg=value*. For instance, the following function:



```
>>> def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...     print("-- Lovely plumage, the", type)
...     print("-- It's", state, "!")
...
>>> parrot
<function parrot at 0x00000207E8E1B700>
>>> parrot()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

Keyword Arguments

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function



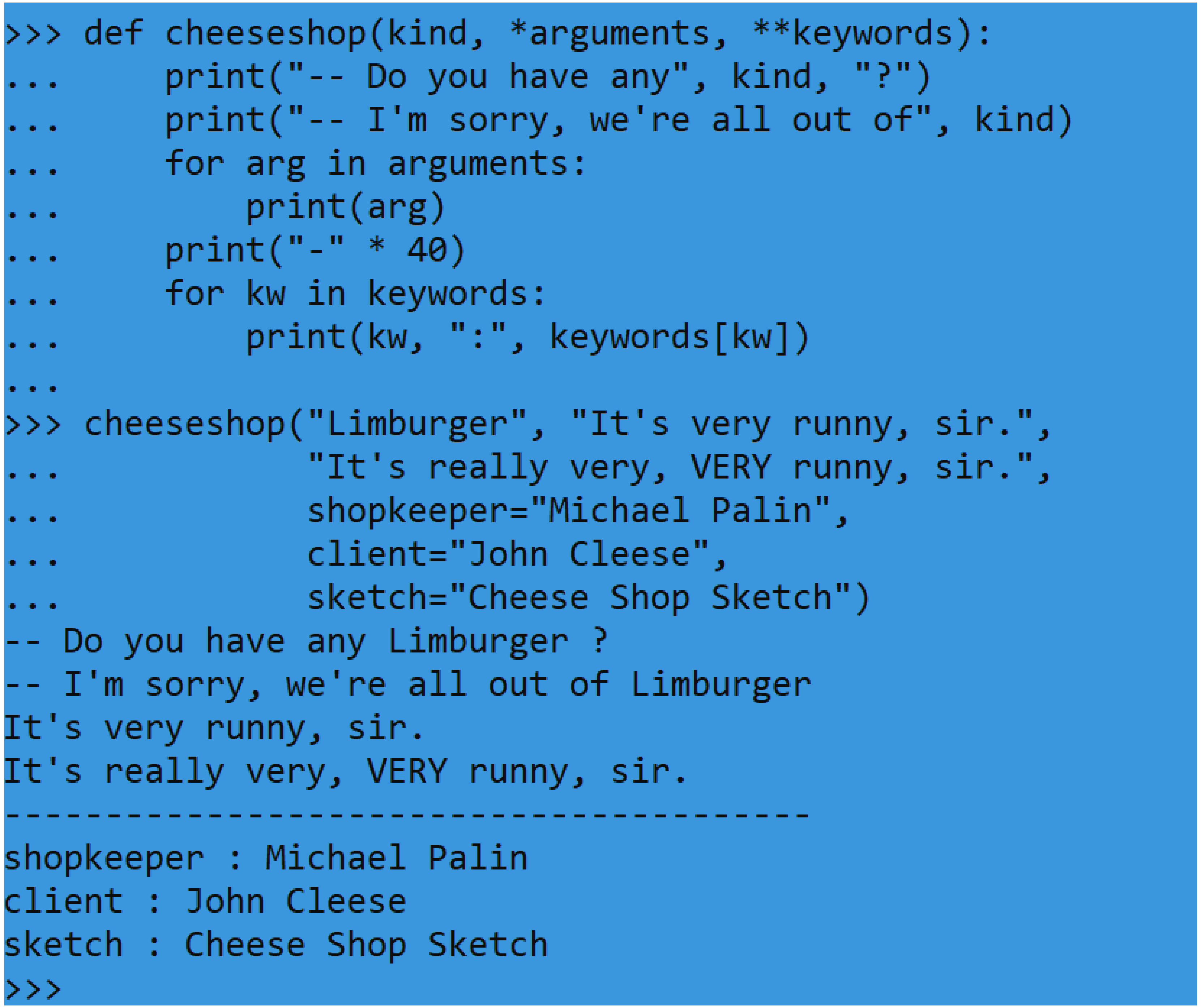
```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
>>>
```

<u>Keyword Arguments</u>

- When a final formal parameter of the ***form **name is present***, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.

- This may be combined with a formal parameter of the form ***name** *which receives a tuple containing the positional arguments beyond the formal parameter list*. ***(*name must occur before **name.)*** For example, if we define a function like this:

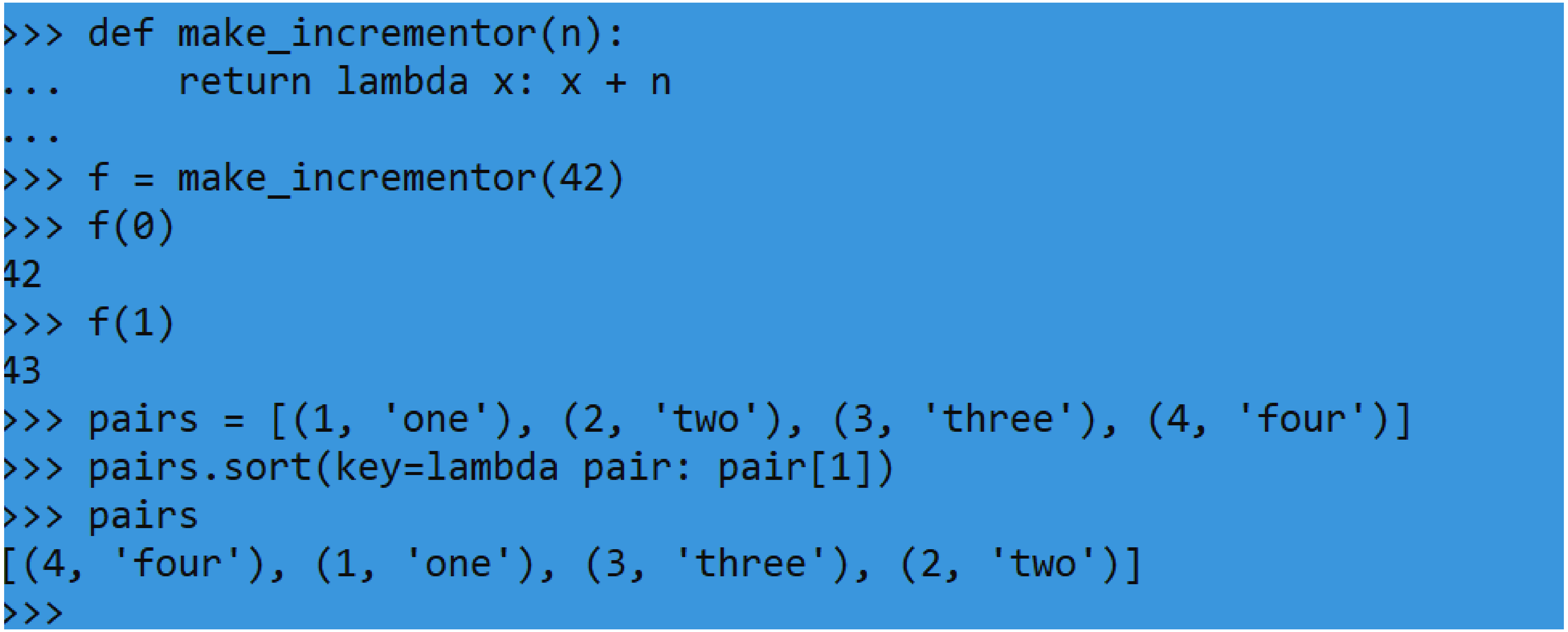
Keyword Arguments



```
>>> def cheeseshop(kind, *arguments, **keywords):
...     print("-- Do you have any", kind, "?")
...     print("-- I'm sorry, we're all out of", kind)
...     for arg in arguments:
...         print(arg)
...     print("-" * 40)
...     for kw in keywords:
...         print(kw, ":", keywords[kw])
...
>>> cheeseshop("Limburger", "It's very runny, sir.",
...            "It's really very, VERY runny, sir.",
...            shopkeeper="Michael Palin",
...            client="John Cleese",
...            sketch="Cheese Shop Sketch")
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
>>>
```

Lambda Expressions

- Small ***anonymous functions*** can be created with the **lambda keyword**.
- This function returns the sum of its two arguments: lambda a, b: a+b.
- Lambda functions can be used wherever function objects are required.
- They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition.
- Like nested function definitions, lambda functions can reference variables from the containing scope:



```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
>>>
```

Data Structures

More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

list.append(x)
Add an item to the end of the list. Equivalent to ***a[len(a):] = [x].***

list.extend(iterable)
Extend the list by appending all the items from the iterable. Equivalent to **a[len(a):] = iterable**.

list.insert(i, x)
Insert an item at a given position. The first argument is the index of the element before which to insert, so **a.insert(0, x)** inserts at the front of the list, ***and a.insert(len(a), x) is equivalent to a.append(x).***

Data Structures

list.remove(x)
Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item.

list.pop([i])
Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.

list.clear()
Remove all items from the list. Equivalent to del a[:].

list.index(x[, start[, end]])
Return the zero-based index in the list of the first item whose value is equal to x. Raises a ValueError if there is no such item.

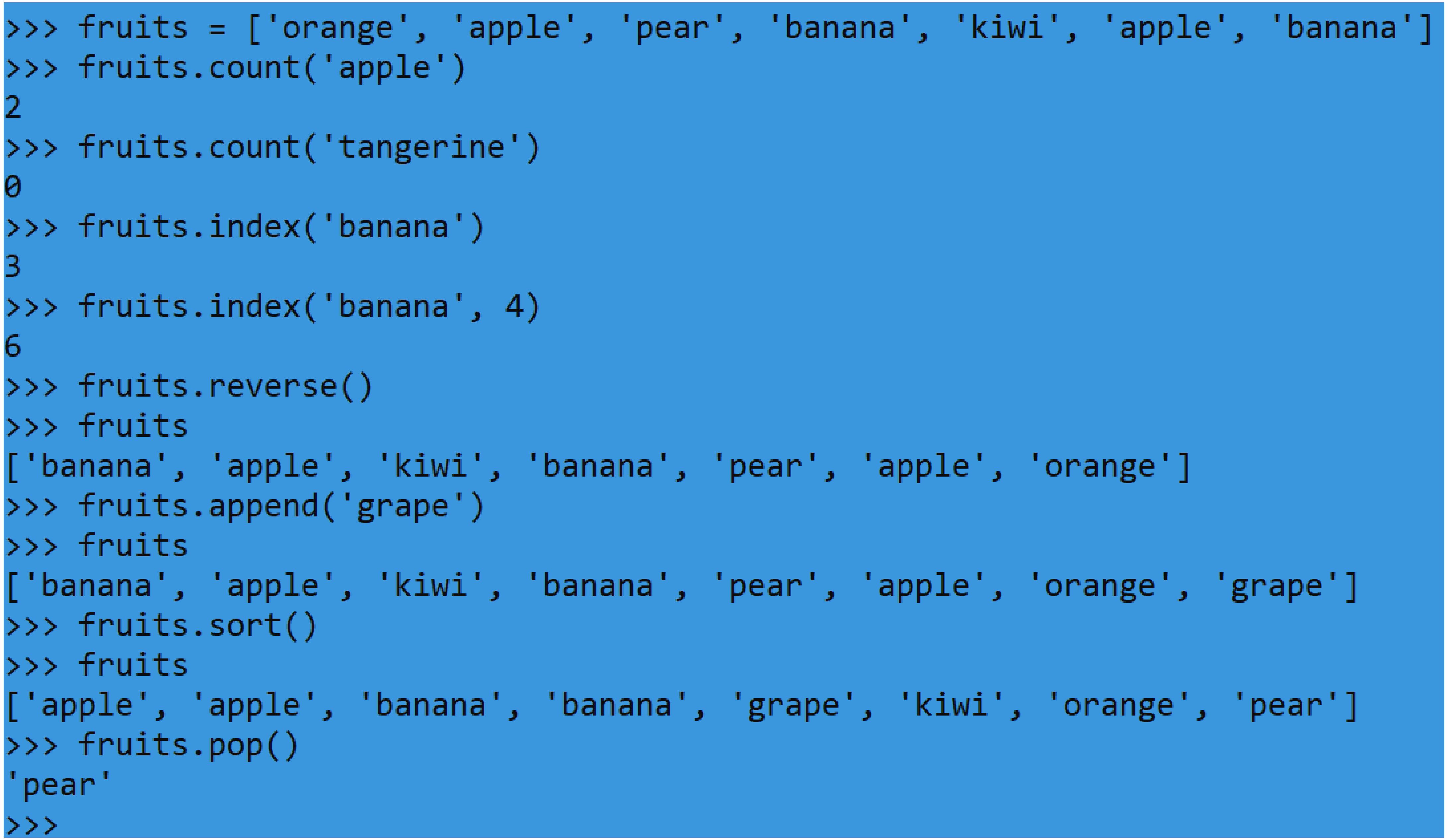
list.count(x)
Return the number of times x appears in the list.

Data Structures

list.sort(, key=None, reverse=False)*
Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

list.reverse()
Reverse the elements of the list in place.

list.copy()
Return a shallow copy of the list. Equivalent to a[:].



```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
>>>
```