New topic

# Modules and Functions

We have already seen that Python provides built-in function that we can use:

```
>>>
>>> len('abcdef')
6
>>> float(5)
5.0
>>> int(5.8)
5
>>> int('125')
125
>>>
```

These functions are available directly from the interactive shell. But …. say we want to get the square root of a number. It would seem reasonable that Python would provide a function to do that as well.

But when I try to use what I think should work, I get this:

```
>>>
>>> sqrt(9)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    sqrt(9)
NameError: name 'sqrt' is not defined
>>>
```

But the following will work.

```
>>> import math
>>> math.sqrt(9)
3.0
>>>
```

**Why**?

"**math**" is a **module (= a file)** containing a number of mathematical functions.

The "**import**" statement instructs Python to "load" the module and make these functions available for use.

**Which functions are available in the math module?**

We do it with the “dir” command:

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'er
fc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gam
ma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', '
log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 't
runc']
>>>
```

If we want to know what each one of these function does we use the “help” command:

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the hyperbolic arc sine (measured in radians) of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
```

And it goes on and on ..

You can also get help on a **single function**:

```
>>> help(math.tan)
Help on built-in function tan in module math:

tan(...)
    tan(x)

    Return the tangent of x (measured in radians).
```

**How do we use functions in a module?**

**There are <u>three</u> ways.**

1. The way we just saw: This just makes the module available but we need to use the "dot" syntax to actually access the function.

```
>>> import math
>>> math.sqrt(9)
3.0
>>>
```

2. We can **<u>import a single function</u>** from a module. The function is then available to be used "directly" like the len() function.

```
>>> from math import sqrt
>>> sqrt(9)
3.0
>>>
```

3. We can **<u>import all the functions from a module at one time</u>**. We can then use the function name directly as in 2 above.

```
>>> from math import *
>>> sqrt(9)
3.0
>>>
```

Question:

What are the advantages and disadvantages of each of the methods above?

Answer:

We can also do this:

```
>>>
>>> import math
>>> sqrt=math.sqrt
>>> sqrt(9)
3.0
>>>
```


## We can create <u>our own functions and modules</u>!

But why would we want to? There are a number of reasons.

Answer:

How do we create functions? Easy. We use "**def**:

```
>>>
>>> def gt(x,y):
        if x>y:
            return True
        else:
            return False

>>> gt(3,4)
False
>>> gt(4,3)
True
>>>
```

**<u>IMPORTANT:</u>** **<u>Terminology</u>**: The x any above are called **<u>parameters</u>**, the 3 and 4 are called **<u>arguments</u>**. The arguments can be constants as in the above example, or variables as in the examples below.

Here is the **syntax**.

**def function_name( parameter list): # the parameter list could be empty – but still need ().**

**code block**

And the **semantics**:

1. A function needs to be defined, using the "def" construction above, before it is used. Otherwise Python will issue an error like this:

```
>>>
>>> is_prime(5)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    is_prime(5)
NameError: name 'is_prime' is not defined
>>>
```

2. The parameter list is a list of variables that will refer to local copies of the arguments "**passed**" from the calling code. For example:

```
>>>
>>> def add(x,y):
        x+=1
        y+=1
        print(x,y)

>>> a=10
>>> b=20
>>> add(a,b)
11 21
>>> a
10
>>> b
20
>>>
```

**Notice1**: The variables in the parameter list are **local**. This means that any changes that you make to them in the function do not affect the values in the corresponding arguments in the main program. What happens in Vegas ….

**Notice2**: The actual story is a bit more subtle than **Notice1**. We have passed in simple types. Stay tuned for what happens later on.

**Question: How does a function return a value back to the "caller"?**

**Answer: It uses the "return statement". It has two forms:**

- **return <some value>**
- **return**
- **or … the function just "falls off" the last statement and implicitly returns to the caller.**

The first form terminates the function and makes <value> available at the place from which the function was called. Program execution resumes from that point as well.

The second form just terminates the function, and computation resumes from the point from which the program was called.

The third "form" behaves just like the one above.

**Terminology:** When a function doesn't return a value, but rather performs some function for us, we will sometimes call it a **procedure**.

```
>>> type(len('asd'))
<class 'int'>
>>>
```

This tells us that the len() function returns an int.

```
>>> type(print('asd'))
asd
<class 'NoneType'>
>>>
```

But the print function returns "NoneType", a catchall that says that the function returns nothing. The print function is not computing a value for us, it's basically "doing a job" for us, and then returning to the caller. We will call this a **procedure**.

**Problem:**

Write a function is_even(x) which returns True if x is even and False otherwise.

Answer:

Problem:

Write a function is_leap(x) which returns True if x is a leap year and False otherwise.Answer:

Problem:

Write a function is_prime(x) which returns True if x is a prime number and False otherwise.

Answer:

Problem:

Write a program to ask the user for two integers, first and last. Write a program to print out all the primes between first and last (inclusive), five values per line.

Answer:

Problem:

Write a function sum_of_digits(n) which returns the sum of the digits of n.

Answer:

Just like there are conversions, int, float, str, there is a built-in function called **bin**. The documentation says:

`bin`(*x*)

Convert an integer number to a binary string.

For example:

```
>>>
>>> bin(6)
'0b110'
>>>
```

**Problem**:

Write a function my_bin(n) which converts an integer number to a binary string representation of n. But Leave out the leading '0b' returned by the built in function bin.

## Variable scoping in Python

The term scoping refers to the visibility of variables (and **all** names) from within the program. If I set a variable's value within a function, have I affected it outside of the function as well? What if I set a variable's value inside a for loop?

**Python has four levels of scoping:**

- Local (i.e. the function that you are in)
- Enclosing function
- Global
- Built-ins

These are known by the abbreviation LEGB.

These are known by the abbreviation LEGB. If you're in a function, then all four are searched, in order. If you're outside of a function, then only the final two (globals and built-ins) are searched. Once the identifier is found, Python stops searching.

That's an important consideration to keep in mind. If you haven't defined a function, you're operating at the global level. Indentation might be pervasive in Python, but it doesn't affect variable scoping at all.

But what if you run int('s')? Is int a global variable? No, it's in the built-ins namespace. Python has very few reserved words; many of the most common types and functions we run are neither globals nor reserved keywords. Python searches the builtins namespace after the global one, before giving up on you and raising an exception.

What if you define a global name that's identical to one in built-ins?
Then you have effectively shadowed (i.e. hidden) the "higher" value.

```
sum = 0
for i in range(5):
   sum += i
print(sum)

print(sum([10, 20, 30]))  # sum is a built-in that accepts any appropriate iterable
```

TypeError: 'int' object is not callable

Why do we get this weird error?
Because in addition to the sum function defined in built-ins, we have now defined a global variable named sum. And because globals come before built-ins in Python's search path, Python discovers that sum is an integer and refuses to invoke it.

It's a bit frustrating that the language doesn't bother to check or warn you about redefining names in built-ins. However, there are tools (e.g., pylint) that will tell you if you've accidentally (or not) created a clashing name.

LOCAL VARIABLES

Firstly, function parameters have local scope, so that any changes to them do not affect their "original" value outside the function. This is loke call by value in C++.

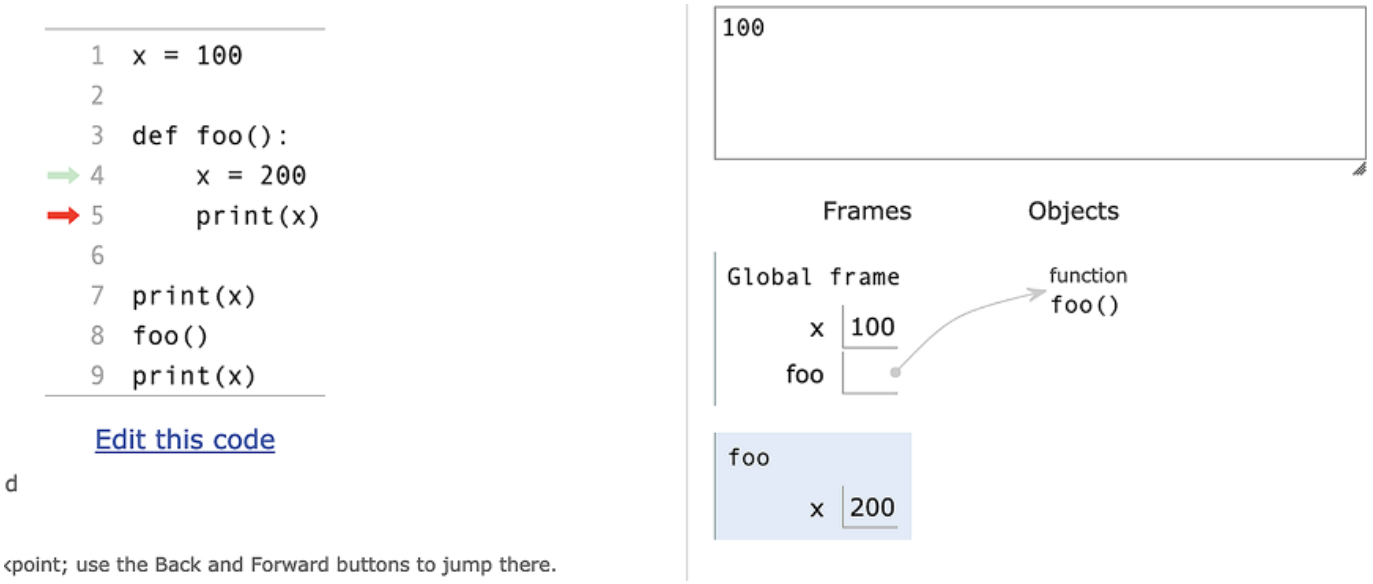
If I define a variable inside a function, then it's considered to be a local variable. Local variables exist only as long as the function does; when the function goes away, so do the local variables it defined; for example

```
x = 100

def foo():
    x = 200
    print(x)

print(x)
foo()
print(x)
```

This code will print 100, 200, and then 100 again. In the code, we've defined two variables: x in the global scope is defined to be 100 and never changes, whereas x in the local scope, available only within the function foo, is 200 and never changes. The fact that both are called x doesn't confuse Python, because from within the function, it'll see the local x and ignore the global one entirely.




d

‹point; use the Back and Forward buttons to jump there.

THE GLOBAL STATEMENT

What if, from within the function, I want to change the global variable?

Use the global statement.

```
x = 100

def foo():
    global x
```

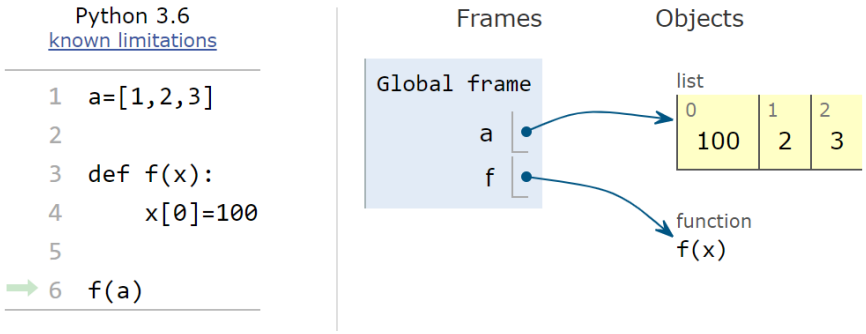
```
    x = 200
    print(x)

print(x)
foo()
print(x)
```

This code will print 100, 200, and then 200, because there's only one x, thanks to the global declaration.

This behavior changes when we pass mutable iterables to a function though.

**The basic idea** is that Python utilizes a system, which is known as "Call by Object Reference" or "Call by assignment" (what is usually called call by value). In the event that you pass arguments like whole numbers, strings or tuples to a function, the passing is like call-by-value because you can not change the value of the immutable objects being passed to the function. Whereas passing mutable objects can be considered as call by reference because when their values are changed inside the function, then it will also be reflected outside the function.




**ENCLOSING** (a function defined in inside another)

Unlike other languages, Python allows functions to be defined inside other functions (we will deal with lambdas later). We will encounter this when we look at "decorators."

```
def foo(x):
    def bar(y):
        return x * y
    return bar

f = foo(10)
print(f(20))
```

What are we doing defining bar inside of foo? This inner function and its accessible variables etc. sometimes known as a closure, is a function that's defined when foo is executed. Indeed, every time that we run foo, we get a new function named bar back. The name bar is a local name inside of foo.

When we run the code, the result is 200. It makes sense that when we invoke f, we're executing bar, which was returned by foo. And we can understand how bar has access to y, since it's a local variable. But what about x? How does the function bar have access to x, a local variable in foo?

The answer is **LEGB:**

First, Python looks for x **locally**, in the local function bar.

Next, Python looks for x in the **enclosing function** foo.

If x were not in foo, then Python would continue looking at the **global** level.

And if x were not a global variable, then Python would look in the **built-ins namespace**.

What if I want to change the value of x, a local variable in the enclosing function? It's not global, so the global declaration won't work. In Python 3, though, we have the **nonlocal** keyword. This keyword tells Python: "Any assignment we do to this variable should go to the outer function, not to a (new) local variable";

For example

```
def foo():
    call_counter = 0                          ❶
    def bar(y):
        nonlocal call_counter                 ❷
        call_counter += 1                     ❸
        return f'y = {y}, call_counter = {call_counter}'
    return bar

b = foo()
for i in range(10, 100, 10):                  ❹
    print(b(i))
                          ❺
```

❶ Initializes call_counter as a local variable in foo

❷ Tells bar that assignments to call_counter should affect the enclosing variable in foo

❸ Increments call_counter, whose value sticks around across runs of bar

❹ Iterates over the numbers 10, 20, 30, ... 90

❺ Calls b with each of the numbers in that range

The output from this code is

```
y = 10, call_counter = 1
y = 20, call_counter = 2
y = 30, call_counter = 3
y = 40, call_counter = 4
y = 50, call_counter = 5
y = 60, call_counter = 6
y = 70, call_counter = 7
y = 80, call_counter = 8
y = 90, call_counter = 9
```

Takeaway: the LEGB scoping rule and how it's always, without exception, used to find all identifiers, including data, functions, classes, and modules.

# A deeper dive into Python functions

1. Argument evaluation
2. Default arguments
3. Variadic arguments
4. Keyword arguments
5. Variadic Keyword Arguments
6. Functions Accepting All Inputs
7. Positional-Only Arguments
8. Names, Documentation Strings, and Type Hints
9. Function Application and Parameter Passing
10. Return Values

## 1.Arguments are fully evaluated left-to-right before executing the function body.

```
def add(x, y):
    return x + y
```

For example, add(1+1, 2+2) is first reduced to add(2, 4) before calling the function. This is known as applicative evaluation order. The order and number of arguments must match the parameters given in the function definition. If a mismatch exists, a TypeError exception is raised. The structure of calling a function (such as the number of required arguments) is known as the function's call signature.

## 2.Python allows for default arguments

You can attach default values to function parameters by assigning values in the function definition. For example:

```
def split(line, delimiter=','):
    statements
```

**example:** We have already encountered this in the print function

**Important rules**

1. When a function defines a parameter with a default value, that parameter and all the parameters that follow it are optional.

2. It is not possible to specify a parameter with no default value after any parameter with a default value.

**Default parameter values are evaluated once when the function is first defined,** not each time the function is called. This often leads to surprising behavior if **mutable objects are used as a default since they will retain the change and the original default won't be the "default" anymore**:

```
def f(x, items=[]):
   items.append(x)
   return items

f(1)      # returns [1]
f(2)      # returns [1, 2]
f(3)      # returns [1, 2, 3]
```

Notice how the default argument retains the modifications made from previous invocations and doesn't reinitialize to the empty list. To prevent this, it is better to use None and add a check as follows:

```
def func(x, items=None):
    if items is None:
       items = []
    items.append(x)
    return items
```

**The takeaway**: Never use a mutable value, such as a list or dictionary, as a parameter's default value. You shouldn't do so because default values are stored and reused across calls to the function. This means that if you modify the default value in one call, that modification will be visible in the next call.

So, as a general practice, to avoid such surprises, only use immutable objects for default argument values—numbers, strings, Booleans, None, and so on.

## 3.Variadic Arguments

A function can accept a variable number of arguments if an asterisk (*) is used as a prefix on the **last parameter name**. For example:

```
def product(first, *args):
   result = first
   for x in args:  # note args is an iterable. Its type is <class 'tuple'>.
      result = result * x
   return result

product(10, 20)      # -> 200
product(2, 3, 4, 5)  # -> 120
```

In this case, all of the extra arguments are placed into the args variable as a tuple. You can then work with the arguments using the standard sequence operations—iteration, slicing, unpacking, and so on.

## 3.Keyword Arguments vs Positional Arguments

When calling a function, arguments can be supplied by explicitly naming each parameter and specifying a value. These are known as keyword arguments. Here is an example:

```
def func(w, x, y, z):
    statements

# Keyword argument invocation
func(x=3, y=22, w='hello', z=[1, 2])
```

With keyword arguments, the order of the arguments doesn't matter as long as each required parameter gets a single value.

If you omit any of the required arguments or if the name of a keyword doesn't match any of the parameter names in the function definition, a TypeError exception is raised.

Keyword arguments are evaluated in the same order as they are specified in the function call.

**Important**

Positional arguments and keyword arguments can appear in the same function call, provided that

- all the positional arguments appear first,
- values are provided for all nonoptional arguments, and
- no argument receives more than one value.

Here's an example:

```
def func(w, x, y, z):
    statements

func('hello', 3, z=[1, 2], y=22)  # note z before y, but that's OK
func(3, 22, w='hello', z=[1, 2])    # TypeError. Multiple values for w
```

**We can force the use of keyword arguments**

It is possible to **force the use of keyword arguments**. This is done by listing parameters after a * argument or just by including a single * in the definition.

Consider the following examples:

```
def read_data(filename, *, debug=False):
    ...

def product(first, *values, scale=1):
    result = first * scale
    for val in values:
        result = result * val
```

```
    return result
```

In this example, the debug argument to read_data() can only be specified by keyword. This restriction often improves code readability:

```
data = read_data('Data.csv', True)        # NO. TypeError
data = read_data('Data.csv', debug=True)  # Yes.
```

The product() function takes any number of positional arguments and an optional keyword-only argument. For example:

```
result = product(2,3,4)            # Result = 24
result = product(2,3,4, scale=10)   # Result = 240
```

## 4.Variadic Keyword Arguments

If the last argument of a function definition is prefixed with **, all the additional keyword arguments (those that don't match any of the other parameter names) are placed in a dictionary and passed to the function. The order of items in this dictionary is guaranteed to match the order in which keyword arguments were provided.

**Why do this?**

Arbitrary keyword arguments might be useful for defining functions that accept a large number of potentially open-ended configuration options that would be too unwieldy to list as parameters. Here's an example:

```
def make_table(data, **parms):
    # Get configuration parameters from parms (a dict)
    fgcolor = parms.pop('fgcolor', 'black')
    bgcolor = parms.pop('bgcolor', 'white')
    width = parms.pop('width', None)
    ...
    # No more options
    if parms:
        raise TypeError(f'Unsupported configuration options {list(parms)}')

make_table(items, fgcolor='black', bgcolor='white', border=1,
                  borderstyle='grooved', cellpadding=10,
                  width=400)
```

The pop() method of a dictionary removes an item from a dictionary, returning a possible default value if it's not defined. The parms.pop('fgcolor', 'black') expression used in this code mimics the behavior of a keyword argument specified with a default value.

## 5.Functions Accepting All Inputs

**By using both * and **,** you can write a function that accepts any combination of arguments. The positional arguments are passed as a tuple and the keyword arguments are passed as a dictionary. For example:

```
# Accept variable number of positional or keyword arguments
def func(*args, **kwargs):
    # args is a tuple of positional args
    # kwargs is dictionary of keyword args
    ...
```

This combined use of *args and **kwargs is commonly used to write wrappers, decorators, proxies, and similar functions. For example, suppose you have a function to parse lines of text taken from an iterable:

```
def parse_lines(lines, separator=',', types=(), debug=False):
    for line in lines:
        ...
        statements
        ...
```

Now, suppose you want to make a special-case function that parses data from a file specified by filename instead. To do that, you could write:

```
def parse_file(filename, *args, **kwargs):
    with open(filename, 'rt') as file:
        return parse_lines(file, *args, **kwargs)
```

The benefit of this approach is that the parse_file() function doesn't need to know anything about the arguments of parse_lines(). It accepts any extra arguments the caller provides and passes them along. This also simplifies the maintenance of the parse_file() function. For example, if new arguments are added to parse_lines(), those arguments will magically work with the parse_file() function too.

## 6.Positional-Only Arguments

Many of Python's built-in functions only accept arguments by position. You'll see this indicated by the presence of a slash (/) in the calling signature of a function shown by various help utilities and IDEs. For example, you might see something like func(x, y, /). This means that all arguments appearing before the slash can only be specified by position. Thus, you could call the function as func(2, 3) but not as func(x=2, y=3). For completeness, this syntax may also be used when defining functions. For example, you can write the following:

```
def func(x, y, /):
    pass

func(1, 2)     # Ok
func(1, y=2)   # Error
```

This definition is supported only in Python 3.8 and later. However, it can be a useful way to avoid potential name clashes between argument names. For example, consider the following code:

```
import time

def after(seconds, func, /, *args, **kwargs):
    time.sleep(seconds)
    return func(*args, **kwargs)


def duration(*, seconds, minutes, hours):
    return seconds + 60 * minutes + 3600 * hours

after(5, duration, seconds=20, minutes=3, hours=2)
```

In this code, seconds is being passed as a keyword argument, but it's intended to be used with the duration function that's passed to after(). The use of positional-only arguments in after() prevents a name clash with the seconds argument that appears first.

## 7.Names, Documentation Strings, and Type Hints

The standard naming convention for functions is to use lowercase letters with an underscore ( _ ) used as a word separator—for example, read_data() and not readData(). If a function is not meant to be used directly because it's a helper or some kind of internal implementation detail, its name usually has a single underscore prepended to it—for example, _helper(). These are only conventions, however. You are free to name a function whatever you want as long as the name is a valid identifier.

The name of a function can be obtained via the __name__ attribute. This is sometimes useful for debugging.

```
>>> def square(x):
...     return x * x
...
>>> square.__name__
'square'
>>>
```

**It is common for the first statement of a function to be a documentation string describing its usage.**

For example:

```
def factorial(n):
    '''
    Computes n factorial. For example:

    >>> factorial(6)
    120
    >>>
    '''
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

The documentation string is stored in the __doc__ attribute of the function. It's often accessed by IDEs to provide interactive help.

**Functions can also be annotated with type hints. For example:**

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

The type hints don't change anything about how the function evaluates. That is, the presence of hints provides no performance benefits or extra runtime error checking. The hints are merely stored in the __annotations__ attribute of the function which is a dictionary mapping argument names to the supplied hints. Third-party tools such as IDEs and code checkers might use the hints for various purposes.

Sometimes you will see type hints attached to local variables within a function. For example:

Click here to view code image

```
def factorial(n:int) -> int:
    result: int = 1       # Type hinted local variable
    while n > 1:
        result *= n
        n -= 1
    return result
```

**Such hints are completely ignored by the interpreter**. They're not checked, stored, or even evaluated. Again, the purpose of the hints is to help third-party code-checking tools.

Adding type hints to functions is not advised unless you are actively using code-checking tools that make use of them. It is easy to specify type hints incorrectly—and, unless you're using a tool that checks them, errors will go undiscovered until someone else decides to run a type-checking tool on your code.

## 8.Function Application and Parameter Passing

When a function is called, the function parameters are local names that get bound to the passed input objects. Python passes the supplied objects to the function "as is" without any extra copying.

**Care is required if mutable objects, such as lists or dictionaries, are passed.** If changes are made, those changes are reflected in the original object. Here's an example:

```
def square(items):
    for i, x in enumerate(items):
        items[i] = x * x    # Modify items in-place
a = [1, 2, 3, 4, 5]
square(a)        # Changes a to [1, 4, 9, 16, 25]
```

Functions that mutate their input values, or change the state of other parts of the program behind the scenes, are said to have "side effects."

As a general rule, side effects are best avoided. They can become a source of subtle programming errors as programs grow in size and complexity—it may not be obvious from reading a function call if a function has side effects or not. Such functions also interact poorly with programs involving threads and concurrency since side effects typically need to be protected by locks.

**It's important to make a distinction between modifying an object and reassigning a variable name. Consider this function:**

```
def sum_squares(items):
    items = [x*x for x in items]    # Reassign "items" name
    return sum(items)

a = [1, 2, 3, 4, 5]
result = sum_squares(a)
print(a)        # [1, 2, 3, 4, 5]    (Unchanged)
```

In this example, it appears as if the sum_squares() function might be overwriting the passed items variable. Yes, the local items label is reassigned to a new value. But the original input value (a) is not changed by that operation. Instead, the local variable name items is bound to a completely different object—the result of the internal list comprehension. There is a difference between assigning a variable name and modifying an object. When you assign a value to a name, you're not overwriting the object that was already there—you're just reassigning the name to a different object.

**Stylistically, it is common for functions with side effects to return None as a result. As**

**an example, consider the sort() method of a list:**

```
>>> items = [10, 3, 2, 9, 5]
>>> items.sort()      # Observe: no return value
>>> items
[2, 3, 5, 9, 10]
>>>
```

The sort() method performs an in-place sort of list items. It returns no result. The lack of a result is a strong indicator of a side effect—in this case, the elements of the list got rearranged.

Sometimes you already have data in a sequence or a mapping that you'd like to pass to a function. To do this, you can use * and ** in function invocations.

For example:

```
def func(x, y, z):
    ...

s = (1, 2, 3)
# Pass a sequence as arguments
result = func(*s)

# Pass a mapping as keyword arguments
d = { 'x':1, 'y':2, 'z':3 }
result = func(**d)
```

You may be taking data from multiple sources or even supplying some of the arguments explicitly, and it will all work as long as the function gets all of its required arguments, there is no duplication, and everything in its calling signature aligns properly. You can even use * and ** more than once in the same function call. If you're missing an argument or specify duplicate values for an argument, you'll get an error. Python will never let you call a function with arguments that don't satisfy its signature.

9.Return Values

The return statement returns a value from a function. If no value is specified or you omit the return statement, None is returned. To return multiple values, place them in a tuple:

```
def parse_value(text):
    '''
    Split text of the form name=val into (name, val)
    '''
    parts = text.split('=', 1)
    return (parts[0].strip(), parts[1].strip())
```

Values returned in a tuple can be unpacked to individual variables:

```
name, value = parse_value('url=http://www.python.org')
```

Sometimes named tuples are used as an alternative:

```
from typing import NamedTuple

class ParseResult(NamedTuple):
    name: str
    value: str

def parse_value(text):
    '''
    Split text of the form name=val into (name, val)
    '''
    parts = text.split('=', 1)
    return ParseResult(parts[0].strip(), parts[1].strip())
```

A named tuple works the same way as a normal tuple (you can perform all the same operations and unpacking), but you can also reference the returned values using named attributes:

```
r = parse_value('url=http://www.python.org')
print(r.name, r.value)
```

# Lists, Strings, Tuples, and Other Sequences

**The following 2 pages are for reference.**

**Sequences** represent ordered sets of objects indexed by nonnegative integers and include strings, (including Unicode strings) lists, and tuples. Strings are sequences of characters, and lists and tuples are sequences of arbitrary Python objects. Strings and tuples are immutable; lists allow insertion, deletion, and substitution of elements. All sequences support iteration.

Don't worry! All the strange terms above will be explained below.

**Operations and Methods Applicable to All Sequences**

| Item | Description |
|---|---|
| s[i] | Returns element iof a sequence |
| s[i:j] | Returns a slice |
| s[i:j:stride] | Returns an extended slice |
| len(s) | Number of elements in s |
| min(s) | Minimum value in s |
| max(s) | Maximum value in s |

---

**Operations Applicable to Mutable Sequences**

| Item | Description |
|---|---|
| s[i] = v | Item assignment |
| s[i:j] = t | Slice assignment |
| s[i:j:stride] = t | Extended slice assignment |
| del s[i] | Item deletion |
| del s[i:j] | Slice deletion |
| del s[i:j:stride] | Extended slice deletion |

**Lists are sequences of arbitrary objects**.

**You create a list as follows**:

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:

```
a = names[2] # Returns the third item of the list, "Ann"
names[0] = "Jeff" # Changes the first item to "Jeff"
```

To append new items to the end of a list, use the append() method:

```
names.append("Kate")
```

To insert an item in the list, use the insert() method:
names.insert(2, “Sydney”)

You can extract or reassign a portion of a list by using the **slicing operator**:
b = names[0:2] # Returns [ “Jeff”, “Mark” ]
c = names[2:] # Returns [ “Sydney”, “Ann”, “Phil”, “Kate” ]
names[1] = ‘Jeff’ # Replace the 2nd item in names with ‘Jeff’
names[0:2] = [‘Dave’,’Mark’,’Jeff’] # Replace the first two items of
# the list with the list on the right.

Use the plus (+) operator to **concatenate** lists:
a = [1,2,3] + [4,5] # Result is [1,2,3,4,5]

Lists can contain any kind of Python object, including other lists, as in the following example:

a = [1,”Dave”,3.14, [“Mark”, 7, 9, [100,101]], 10]

Nested lists are accessed as follows:
a[1] # Returns “Dave”
a[3][2] # Returns 9
a[3][3][1] # Returns 101

**List Methods**

| Method | Description |
|---|---|
| list(s) | Converts sto a list. |
| s.append(x) | Appends a new element, x, to the end of s. |
| s.extend(t) | Appends a new list, t, to the end of s. |
| s.count(x) | Counts occurrences of xin s. |
| s.index(x [,start [,stop]]) | Returns the smallest iwhere s[i] ==x. start and stop optionally specify the start- ing and ending index for the search. |
| s.insert(i,x) | Inserts x at index i. |
| s.pop([i]) | Returns the element iand removes it from the list. If i is omitted, the last element is returned. |
| s.remove(x) | Searches for xand removes it from s. |
| s.reverse() | Reverses items of sin place. |
| s.sort([cmpfunc [, keyf [, reverse]]]) | Sorts items of sin place. cmpfuncis a comparison function. keyfis a key function. reverseis a flag that sorts the list in reverse order. |

We will start with

# Lists

A list in Python is a mutable sequence of any time of Python object.

What does this mean??

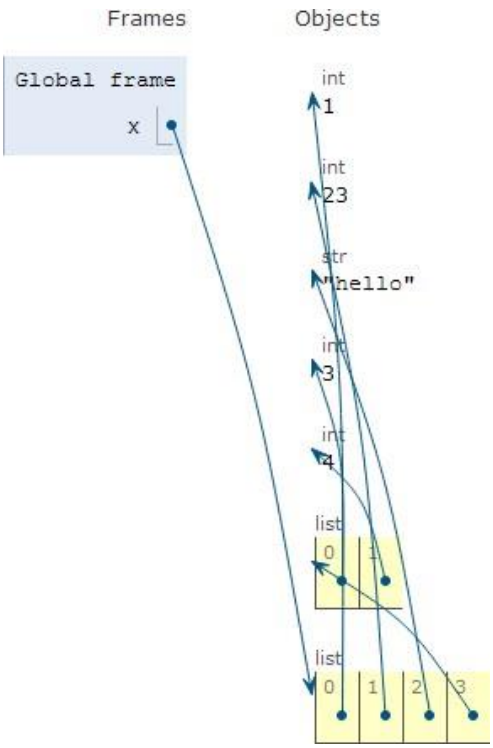
Example:

x=[1,23,”hello” , [3.4]]

x is the name of the list. It has 4 elements:

- the integer 1
- the integer 23
- the string “hello”
- the list [2,3]

It looks something like this in memory.



We create a new empty list in one of two ways:

- x=[]
- x=list()

or, as above, we can create a list with elements just by listing the elements in the square brackets "[" , "]".

**Question**:

How do we **access** individual elements of a list?

**Answer:**

We use **square brackets with an integer** to "index" into the list. For example:

```
>>> x=[1,23,"hello", [3,4]]
>>> print(x[0])
1
>>> print(x[3])
[3, 4]
>>>
>>> print(x[3][0])
3
>>> print(x[4])
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(x[4])
IndexError: list index out of range
>>>
```

Notice:

1. The positions in the list are numbered from 0 (not 1). In the above example, this means that the last element in the lost is accessed as x[3].

2. Since x[3] in our example is the list [3,4] we can access its elements by using a second index. That is why x[3][0] is the "zeroth" (i.e. first) element of [3,4], which is 3.

3. Since there is no element in the list x[4] (they are x[0], x[1], x[2], x[3]) we are trying to access a nonexistent element and Python prints an error message.

**Question**:

Can we change (i.e. replace) elements of a list?

**Answer**:

Yes. Here is an example where we modify the list x above.

```
>>>
>>> x[0]="Bob"
>>> x
['Bob', 23, 'hello', [3, 4]]
>>>
```

We can find out the size (=length) of a loop by using the len() function:

```
>>>
>>> x
['Bob', 23, 'hello', [3, 4]]
>>> len(x)
4
```

We say that a list is **mutable**. This means that is can be modified (i.e. "mutated).

**Question:**

How can we add elements to an existing list?

**Answer:**

There are a number of different ways. We start with two functions:

- append – add "something" to the end of a list
- extend – add all the elements of some **sequence** at the end of a list

```
>>>
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.append("hello")
>>> a
[1, 2, 3, 4, 'hello']
>>> b=[5,6,7]
>>> a.append(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7]]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7]
>>> a.extend(8)
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    a.extend(8)
TypeError: 'int' object is not iterable
>>> a.extend([8])
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7, 8]
>>>
```

Make sure the example above is absolutely clear!

# Lists and loops

List and loops are made for each other!

```
>>>
>>> s=[]
>>> for i in range(1,11):
        s.append(i)

>>> s
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
```

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list and print the sum.

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop, add up all the **elements** of the list **that are even** and print the sum.

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list that are odd and print the sum.

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list **that are in even positions** ( 0 is even) and print the sum.

When using lists it is often convenient to have Python generate some random values for us. Python provides a module called random that has some useful functions for this purpose. The two that we will use most are:

- random and
- randint

```
>>>
>>> from random import random,randint
>>> random()
0.5697709209009185
>>> help(random)
Help on built-in function random:

random(...)
    random() -> x in the interval [0, 1).

>>> randint(3,45)
38
>>> help(randint)
Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

>>>
```

So ..

Function “random()” generates a random **floating point number** from zero up to but not including one.

Function randint(a,b) generates a random **integer** in the range a to b inclusive. Note that a and b here are integers.

Problem:

Write a program to fill a list of size 10 with random integers in the range 1 – 10 and print it out.

Problem:

Modify the program above so that we print the list as well as the maximum integer in the list. Do this two ways.

Problem:

Modify the program above so that it also prints the position in the list where the maximum element was found.

Problem:

Using the code from the program above write a **function**

getmax(x,i) #x is a list and i is an integer

which will find and **return the maximum element among the first i elements of list x**.

getmax will return **two values**:

- the maximum element found, and
- the position in list x where that element was found.

For example, say a=[4,2,7,1,45,23], then getmax(a,4) will search for the maximum element in the first 4 element of list a.

So, in this case it will look at the following numbers: 2,4,7,1, and getmax(a,4) will return 7,2. This because in the first 4 elements, the largest is 7 and it is in position 2.

If we ran getmax(a,6) the function will return 45,4.

Problem:

Generate all the primes between 2 and 100.

Solution:

The **Sieve of Eratosthenes** provides an efficient solution. This algorithm is over **2200 years o**ld!

Here is the Wikipedia description: http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

See there for an animation of the algorithm.

**To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:**

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p, enumerate its multiples by counting to n in increments of p, and mark them in the list (these will be 2p, 3p, 4p, etc.; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

**Why does it work?**

The main idea here is that as we go through the algorithm, every value for p is prime, because we will have already marked all the multiples of the numbers less than p. Note that some of the numbers being marked may have already been marked earlier (e.g. 15 will be marked both for 3 and 5).

**The program below is a slight modification of the above algorithm.**

This program will generate all primes in the range 2-100 using the sieve method.

We use two lists: x and primes.

**list x** will initially have the values 0-100 so that x[i]=i
**list primes,** initially empty, will grow as each new prime is found and appended onto it.

Each time a new prime p in x is located:

- p will be appended to list primes
- its place in list x will be set to zero,
- set all of its multiples in list x will be to zero.

When all elements in x are zero (we check this with the sum function), the main while loop terminates and the program prints list primes.

```
x=[]
for i in range(101):
    x.append(i)

primes=[2]

x[0]=x[1]=x[2]=0

p=2  # the first prime

while sum(x) != 0: # as long as sum(x)!=0, there are still non-zero entries in x.

    # Zero out all multiples of p
    i=1
    while i*p<=100:
        x[p*i]=0
        i=i+1


    # Now, look for the next prime

    p=p+1
    while x[p]==0: # it's at the next non-zero position of x
        p=p+1

    # We found it. Add it to the list of primes, and zero out its position in x

    primes.append(p)
    x[p]=0

# Done! Now print the list of primes.
print(primes)
```

# Let's take a break!

Here is a **really** interesting

## Microsoft/Google/Wall Street Interview Question!

1. Run the following program:

```
from math import sqrt
from random import random

count=0

for i in range(1000000):
    x=random()
    y=random()
    if sqrt(x*x+y*y)<1:
        count+=1

print(4*(count/1000000))
```

2. What is it calculating?

3. How/why does it work? What is the theory behind this?

# Slicing lists

### What is a slice of a list?

If x is a list then **the slice** **x[a:b]** is the "sub-list" of the elements of the elements of a **from** index position a **up to but not including** index position b.

```
>>>
>>> a=[1,2,3,4,5,6,7]
>>> b=a[3:5]
>>> b
[4, 5]
>>>
>>>
```

If we want to indicate that the slice starts at the beginning of the list, we can leave out the start value:

```
>>>
>>> c=a[:5]
>>> c
[1, 2, 3, 4, 5]
>>>
>>>
```

If we want to indicate that the slice goes all the way to the end of the list, we can leave out the end value:

```
>>>
>>> d=a[4:]
>>> d
[5, 6, 7]
>>>
>>>
```

Leaving out both the start and end indexes is the same as saying the whole list. So:

```
>>>
>>> e=a[:]
>>> e
[1, 2, 3, 4, 5, 6, 7]
>>> a
[1, 2, 3, 4, 5, 6, 7]
>>>
>>>
```

**What can we do with a slice of a list?**

1. As we saw above, we can create a new list form a slice.

2. We can assign to a slice and thereby replace one sub-list by another.

```
>>>
>>> a[2:5]=['a','b','c']
>>> a
[1, 2, 'a', 'b', 'c', 6, 7]
>>>
>>>
```

Notice that this is a **generalization** of accessing and replacing one list element as in **a[1]=12** which just replaced a single list element.

**When we use a slice we can indicate a stride.**

**Huh?**

The stride is the length of the "step" that you take going from one element to the next when creating the slice.

In the following example 2 is the stride.

```
>>>
>>> x=[10,20,30,40,50,60,70,80,90]
>>> y=x[1:8:2]
>>> y
[20, 40, 60, 80]
>>>
```

**We can assign a list to a slice with a stride**, **but** the list on the right hand side of the assignment must be the same size as the list produced by the slice. In the following example, both are of size 4.

```
>>>
>>> x[1:8:2]=['a','b','c','d']
>>> x
[10, 'a', 30, 'b', 50, 'c', 70, 'd', 90]
>>>
>>>
```

Note: The right hand side of a slice assignment can be any iterable (a string for example) as long as the lengths are the same.

```
>>> a=[1,2,3,4,5,6,7,8,9,10]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[1:10:2]='abcde'
>>> a
[1, 'a', 3, 'b', 5, 'c', 7, 'd', 9, 'e']
```

# Sorting

**Sorting is an operation on a list that orders the list elements in a specific order.**

For example:

1. **A list of names** may be sorted in "lexical" =dictionary=alphabetical order.

Here is a formal definition of lexical order from Wikipedia:

The name of the lexicographic order comes from its generalizing the order given to words in a dictionary: a sequence of letters (that is, a word)

a1a2 ... ak

appears in a dictionary before a sequence

b1b2 ... bk

if and only if at the first i where ai and bi differ, ai comes before bi in the alphabet.

That comparison assumes both sequences are the same length. To ensure they are the same length, the shorter sequence is usually padded at the end with enough "blanks" (a special symbol that is treated as coming before any other symbol).

Note that we can order the names in reverse order, from latest to earliest. In this case the words still are in lexicographic order, but from the last element to the first.

2. **A list of integers** may be listed in either from smallest to largest or vice versa.

**Why sort?**

It turns out that sorting is one of the most important operations that programs perform. Two examples.

1. **Searching a list**. In order to find a specific element in a list we often sort it first. A sorted list can be searched much more quickly than one that is unsorted. Imagine looking up a phone number in a phone book (list) with a million entries. If the list is unsorted, we might need to look at 1,000,000 entries. If it is sorted, we don't need more than 20.

2. **A Scrabble dictionary**. We might want to bring all the words with the same letters next to each other in the list. So if we got the letters '**opts'** we would like to have stop, pots, and tops all next to one another. Imagine that we had a function called "signature()" that transforms each of stop, tops and post ➔ opts. Then if D is the list of dictionary words then

D.sort(key=signature)

would do this for us. We will actually do this later on.

Problem:

Given a list of integers, sort it so that its elements will be in ascending order.

**Selection Sort**

Here is the beginning of the Wikipedia entry. http://en.wikipedia.org/wiki/Selection_sort

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Here is an example of this sort algorithm sorting five elements:

```
64 25 12 22 11
11 25 12 22 64
11 12 25 22 64
11 12 22 25 64
11 12 22 25 64
```

And here is a simple (but not very efficient) implementation.

It uses two new list functions.

```
a=[4, 2, 7, 1, 45, 23]

def select_sort(x):
    for i in range(len(x)-1):
        y=x[i:] # each time through the loop look for the minimum from position i to the end.
        m=min(y)
        pos=x.index(m,i,len(x)) # find the index of the first element with value m in the range [i,len(x) )
        x[i],x[pos]=x[pos],x[i] # swap the element at position i with the element at position pos

select_sort(a)
print(a)
```

**Notice** that this function uses two list functions **min()** and **index().** In the following, s is a list.

**min(s)** which returns the smallest item of *s*

**s.index(x[, i[, j]])** which return smallest *k* such that s[k] == x and i <= k < j

In the index function i and j are optional. If omitted index searches the whole list. If item x is not found in list s, Python returns an error. In general, we should first as Python “x in s” before using the index function. In function select_sort() we don’t have to do this since we know that m exists.

**Question**: Can you detect two inefficiencies in the implantation above?

Answer:

The following is a more efficient implementation of the same algorithm.

```
def select_sort(x):
    for i in range(len(x)-1):
        m=x[i]
        pos=i
        for j in range(i,len(x)):
            if x[j]<m:
                m=x[j]
                pos=j
        x[i],x[pos]=x[pos],x[i]
```

**Questions:**

Why is it more efficient?

Why does the outer for loop have range(**len(x)-1**) but the inner loop has range(i,**len(x)**)?

Here is **another elementary sort** called

**Bubble Sort**

Here is the beginning of its Wikipedia entry.

**Bubble sort**, sometimes incorrectly referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

The full article and an animation is here: http://en.wikipedia.org/wiki/Bubble_sort

Here is the code.

```
# Bubble Sort

def bubble_sort(a):
    for i in range(len(a)):
        sorted=True
        for j in range(len(a)-i-1):
            if a[j]>a[j+1]:
                a[j],a[j+1]=a[j+1],a[j]
                sorted=False
        if sorted==True:
            return

# Lets test it.
a=[33,6,3,21,88,30]
bubble_sort(a)
print(a)
.
```

**Sorting … a third way**, using Python's **two** built-in sorting functions: **sort()** and **sorted().**

1. **sort()**

Let a be a list.

```
>>>
>>> a.sort(
          L.sort(key=None, reverse=False) -- stable sort *IN PLACE*
```

**Important**: a.sort() will sort the elements in a, thereby changing a.

**Notice**: function sort() takes 2 optional **key word** arguments:

- key
- reverse

**key** specifies a function of one argument that is applied to the list elements before the comparison is made.

**reverse** specifies that the list should be in reverse order. That means that ">" is used for comparison rather than "<".

**Why is it called a "keyword argument"?**

Because if you want to use it, you need to use the "keyword=value" syntax. We have seen keyword arguments before.

Say, **for example**, **we want to sort a list of strings**. String comparison **depends on capitalization** as in the following example. If we wanted to discount the capitalization in the comparison we could use the lower() function.

```
>>> 'abc'<'ABC'
False
>>>
>>>
>>>
>>> 'abc'>'ABC'
True
>>>
>>>
>>> 'abc'.lower()<'ABC'.lower()
False
>>>
>>>
>>> 'ABC'.lower()
'abc'
>>>
```

The default value for key is None.

```
>>>
>>> a=['ONE','two','one','TWO']
>>> b=['ONE','two','one','TWO']
>>> a
['ONE', 'two', 'one', 'TWO']
>>> b
['ONE', 'two', 'one', 'TWO']
>>> a.sort()
>>> a
['ONE', 'TWO', 'one', 'two']
>>> b.sort(key=str.lower)
>>> b
['ONE', 'one', 'two', 'TWO']
>>>
```

What about the keyword argument “reverse”?

Here is an example.

```
>>>
>>> a=[34,4,21,77,5,45,8]
>>>
>>> a.sort()
>>>
>>> a
[4, 5, 8, 21, 34, 45, 77]
>>>
>>>
>>> a.sort(reverse=True)
>>>
>>> a
[77, 45, 34, 21, 8, 5, 4]
>>>
>>>
```

Problem:

Given a list, print the elements of that list in reverse order. Do this in two ways.

Problem:

Given a list reverse the elements of the list. For example if

x=[1,2,3], then after it is reversed x would be [3,2,1].

Do this in two ways.

2. **sorted()**

The sorted function will create a new list.

```
>>> a=[1,2,3]
>>> sorted(
        (iterable, /, *, key=None, reverse=False)
        Return a new list containing all items from the iterable in ascending order.
```

**See the difference between sort() and sorted():**

```
>>> a=[1,2,3]
>>> a.sort(reverse=True)
>>> a
[3, 2, 1]
>>>
>>> a=[1,2,3]
>>> b=sorted(a,reverse=True)
>>> a
[1, 2, 3]
>>> b
[3, 2, 1]
>>>
```

**Note:** The built-in sorted() function is guaranteed to be **stable**.

**Definition**: A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

## We saw the use of the keyword reverse, what about the keyword key?

## Key Functions

As we saw above, both list.sort() and sorted() have a key parameter to specify a function (or other callable) **to be called on each list element** prior to making comparisons. That is, if f() is the key function then instead if comparing elements a and b , f(a) and f(b) are compared instead.

For example, here's a case-insensitive string comparison:

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

In the above example, the function, lower(), is predefined for string objects, but we can use our own functions as well.

The value of the key parameter should be a function (or other callable) that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

In the context of sorting, (and others as we will see later) it is quite common to use a special kind of function called a "lambda expression."

## lambda Expressions

The lambda expression is an anonymous—unnamed—function with the following form:

lambda args: expression

where args is a comma-separated list of arguments, and expression is an expression involving those arguments.

**Here's an example:**

```
a = lambda x, y: x + y
r = a(2, 3)          # r gets 5
```

The code defined with lambda must be a valid expression. Multiple statements, or nonexpression statements such as try and while, **cannot** appear in a lambda expression.

**A common pattern is to sort complex objects using some of the object's indices as keys.**

For example:

```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

**The key-function pattern is very common**, so Python provides convenience functions to make accessor functions easier and faster. The operator module has the itemgetter() (also attrgetter(), and a methodcaller() which we will see later) function.

Using those functions, the above becomes simpler and faster:

```
>>> from operator import itemgetter, attrgetter

>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

**Problem**: Explain the result:

```
>>> a
[1, 2, 3]
>>> a.sort(reverse=True)==sorted(a,reverse=True)
False
>>>
```

# Two dimensional lists

Many important applications use data that is represented in a 2-dimensional table.

| | | |
|---|---|---|
| 10 | 20 | 30 |
| 40 | 50 | 60 |
| 70 | 80 | 90 |

**How do we represent this in Python?**

We simply use a list of lists.

a=[ [10,20,30],[40,50,60],[70,80,90] ]

**Notice** that the length of list a is 3 (len(a)==3), but it's made up of three lists, each one of length 3.

```
>>>
>>> a=[ [10,20,30],[40,50,60],[70,80,90] ]
>>> len(a)
3
>>> len(a[0])
3
>>>
>>>
```

Problem:

Change element with a 50 to 500.

Solution:

The 50 is the second element of the second list. Recalling that lists are indexed starting with 0, we write:

```
>>>
>>>
>>> a[1][1]=500
>>> a
[[10, 20, 30], [40, 500, 60], [70, 80, 90]]
>>>
>>>
```

# Nested loops and two-dimensional lists

Even though a list is "really" is a list of lists, when we program its useful to think of it as a two dimensional table.

So, for the list a above, **we can consider it a table with three rows and three columns**. The rows and columns are each indexed starting at 0. **We will say that he position with entry 500 is at row 1 and column 1.**

We saw how lists and loops are "made for each other." The same is true with two dimensional lists (we will sometimes refer to the as two dimensional "arrays". This is what they are called in many other programming languages (though they are implemented differently).

Problem:

Print list a above so that each "row" of it prints a separate row.

```
>>>
>>> for i in range(3):
        for j in range(3):
                print(a[i][j],end=' ')
        print()

10 20 30
40 500 60
70 80 90
>>>
>>>
```

And formatted …

```
>>>
>>> for i in range(3):
        for j in range(3):
                print(format(a[i][j],">6d"),end=' ')
        print()

    10     20     30
    40    500     60
    70     80     90
>>>
>>>
```

Problem:

Create a 4X4 array and initialize each of the elements to 0.

Solution:

```
a=[]
for i in range(4):
   a.append(4*[0])
```

**What does 4*[8] mean?**

Python lets us use + and * with lists.

```
>>>
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> a+4
Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    a+4
TypeError: can only concatenate list (not "int") to list
>>> a+[4]
[1, 2, 3, 4]
>>>
>>>
```

So from here you see + is **concatenate** i.e. it acts like the list function **extend**. But you can only + a list to a list, not a string to a list like you can with extend.

**What about "*" ?**

```
>>>
>>> a=4*[8]
>>> a
[8, 8, 8, 8]
>>>
>>>
```

**Make sure that you can explain each of the following:**

```
>>> a=3*3*[[0]]
>>> a
[[0], [0], [0], [0], [0], [0], [0], [0], [0]]
>>> a=3*3*[0]
>>> a
[0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> a=3*[3*[0]]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

The first example:

The second example:

The third example:

The fourth example (below). Does this produce the same result as third example above?

```
>>> a=[]
>>> for i in range(3):
        a.append(3*[0])


>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

Notice when we print list a, both seem to produce the same list:

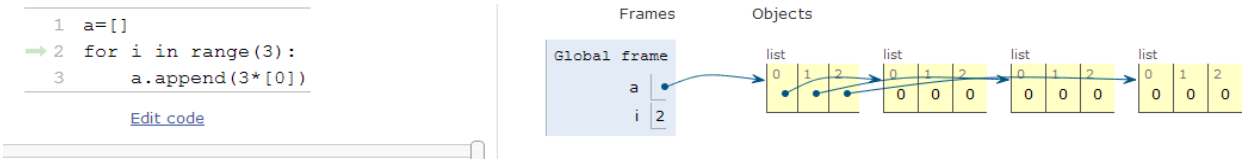
```
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

**However, internally they are represented very differently.**

The first one produces the following when it runs:


What are the implications of this?

The second one, however, produces this:




What are the implications of this?