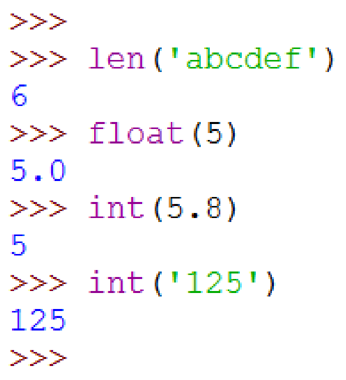
New topic

Modules and Functions

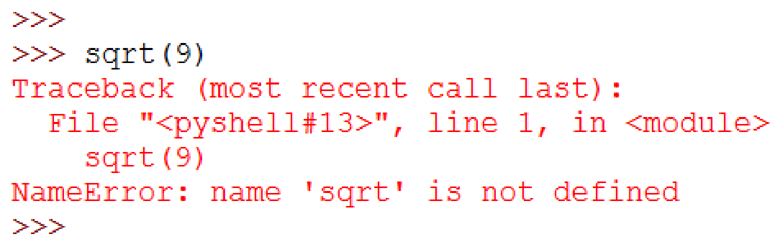
We have already seen that Python provides built-in function that we can use:



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

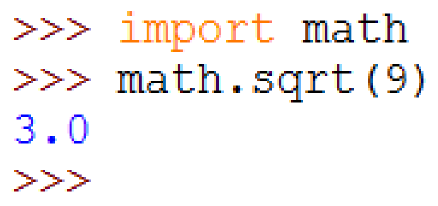
These functions are available directly from the interactive shell. But …. say we want to get the square root of a number. It would seem reasonable that Python would provide a function to do that as well.

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
>>>
```

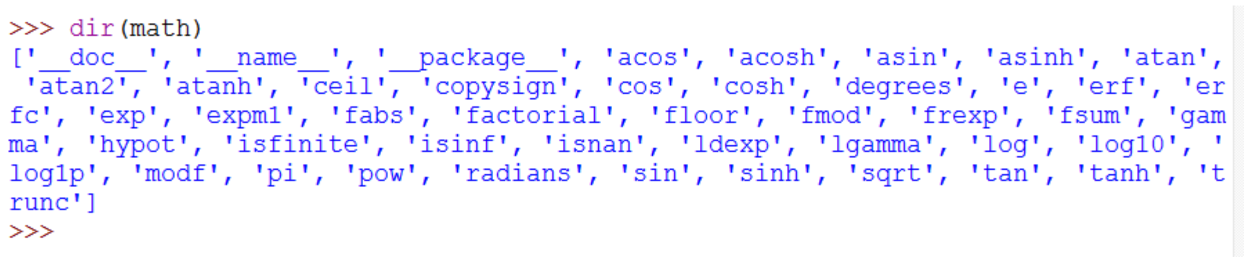
Why?

“**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.

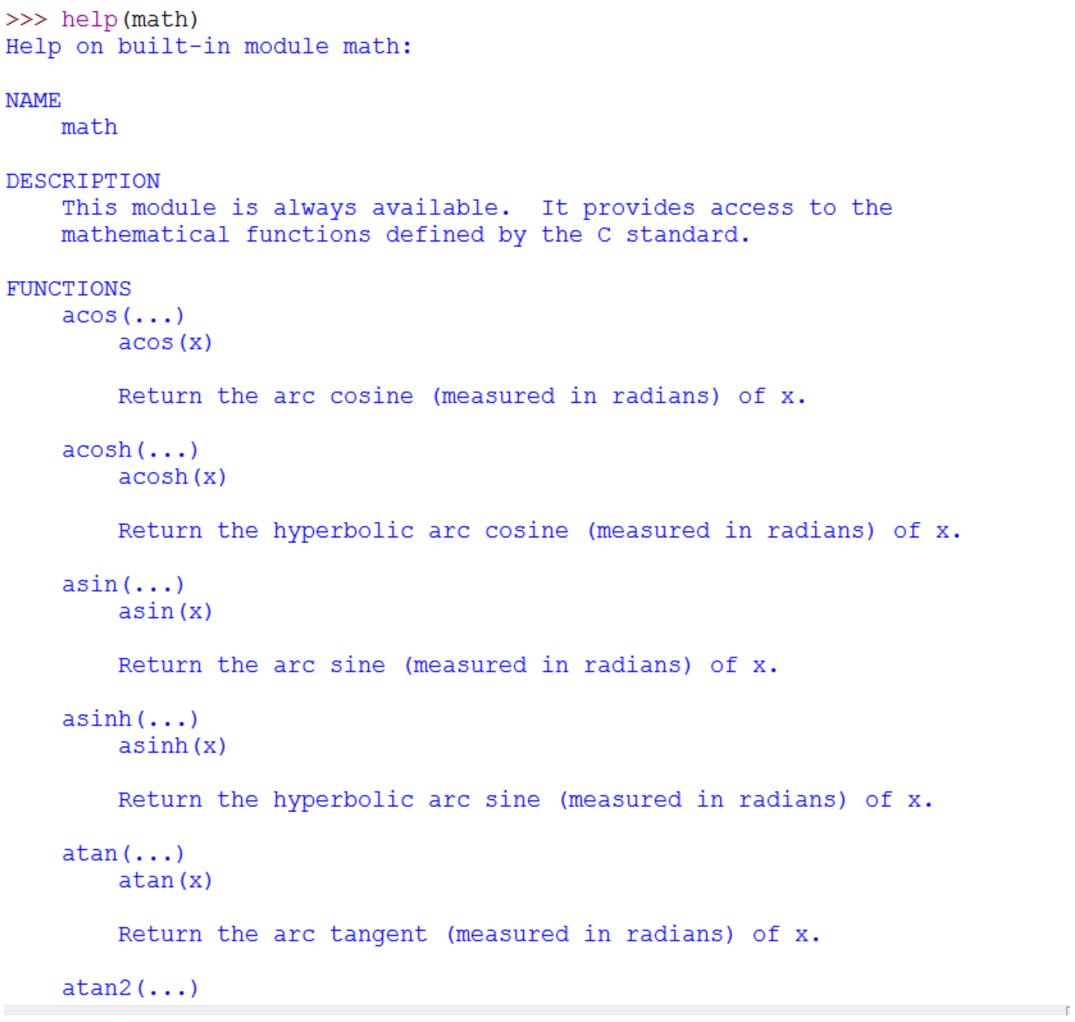
Which functions are available in the math module?

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']
>>>
```

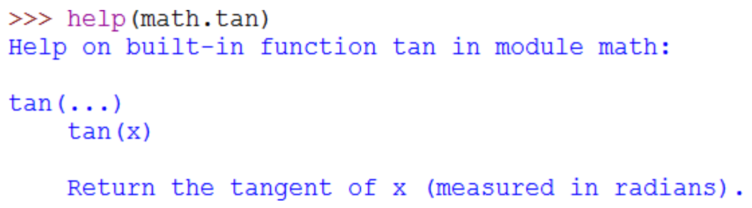
If we want to know what each one of these function does we use the “help” command:



```
>>> 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**:

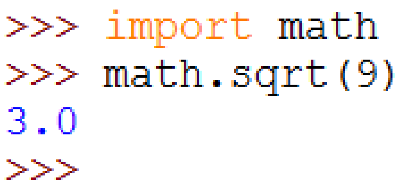


```
>>> help(math.tan)
Help on built-in function tan in module math:

tan(...)
    tan(x)

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

How do we use functions in a module?

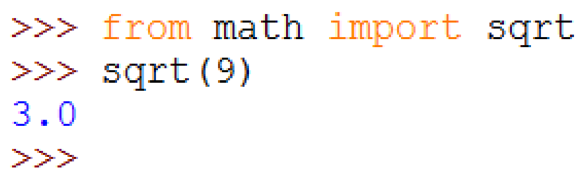
There are <u>three</u> ways.

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.



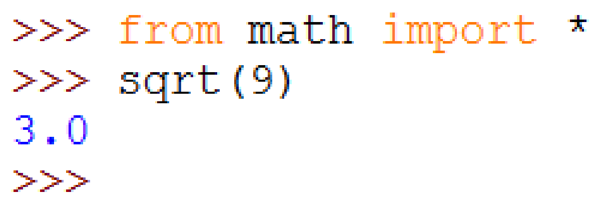
```
>>> import math
>>> math.sqrt(9)
3.0
>>>
```

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.



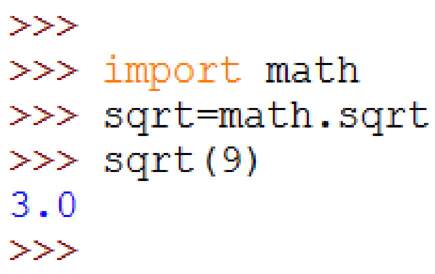
```
>>> from math import *
>>> sqrt(9)
3.0
>>>
```

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
>>>
```

FLASH!!!!

Of course, we can create <u>our own functions and modules</u>!

<u>Overview</u>

Functions are a fundamental building block that allow for the modularization and reusability of code. Understanding how functions are implemented in Python involves several key concepts, including function definition, arguments, return values, scope, and closures.

1. Function Definition

A function in Python is defined using the `def` keyword, followed by the function name and parentheses containing any parameters the function might take. The body of the function starts on the next line and must be indented.

```
def my_function(param1, param2):
    # Function body
    result = param1 + param2
    return result
```

2. Function Arguments

Functions **can** take arguments, which are values passed to the function when it is called. Python supports several types of arguments:

<u>Positional arguments</u>: If you have them, it’s their position in the function call matters. They are like the function arguments in the other languages that you are familiar with.
<u>Keyword arguments</u>: These are optional and have default values. They are identified by the keyword used in the function call, not their position.
<u>Arbitrary positional arguments</u>: If a function needs to accept an arbitrary number of positional arguments, it can use `*args`.
<u>Arbitrary keyword arguments:</u> To accept an arbitrary number of keyword arguments, a function can use `**kwargs`.

```
def example_function(positional, *args, keyword='default', **kwargs):
    pass
```

3. Return Values

Functions in Python can return values using the `return` statement. If no `return` statement is present or if `return` is called without a value, the function will return `None`.

```
def add(a, b):
   return a + b
```

4. Scope and Namespaces

In Python, every function creates its own scope, which is the context in which its variables are defined and accessed. Variables defined inside a function are local to that function and not accessible outside of it. Python uses namespaces to keep track of all the variables and their scopes.

Local scope: Refers to variables defined within a function.
Global scope: Refers to variables defined at the top level of a module or declared global using the `global` keyword within a function.
Enclosing scope: Refers to the scope of any enclosing functions, relevant in the context of nested functions.

5. First-Class Objects

In Python, functions are first-class objects, meaning they can be passed around and used as arguments or return values in other functions. This allows for higher-order functions and functional programming patterns.

```
def shout(text):
   return text.upper()

def whisper(text):
   return text.lower()

def greet(func):
   greeting = func("Hello, I am a function")
   print(greeting)

greet(shout)  # Output: "HELLO, I AM A FUNCTION"
greet(whisper)  # Output: "hello, i am a function"
```

6. Closures

A closure in Python is a function object that remembers values in enclosing scopes even if they are not present in memory. It is a record that stores a function together with an environment: a mapping associating each free variable of the function with the value or reference to which the name was bound when the closure was created.

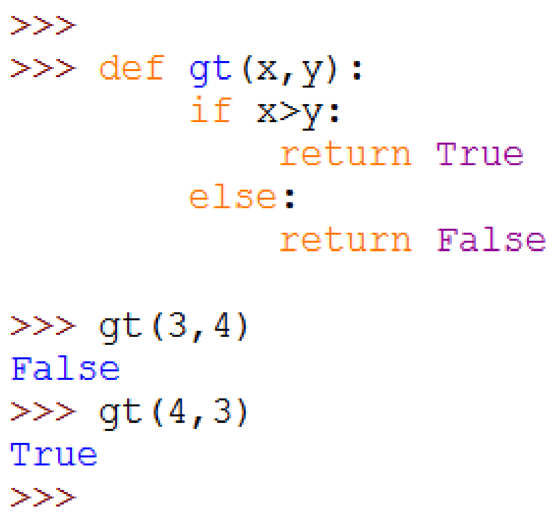
```
def outer_function(text):
    def inner_function():
        print(text)
    return inner_function  # Return the inner function

my_func = outer_function('Hello')
my_func()  # Output: "Hello"
```

7. Decorators

Decorators are a powerful aspect of Python functions, allowing you to modify the behavior of a function without changing its code. A decorator is a function that takes another function as an argument, wraps its behavior in an inner function, and returns the wrapped function.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Let's see/do some examples.



```
>>>
>>> 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.

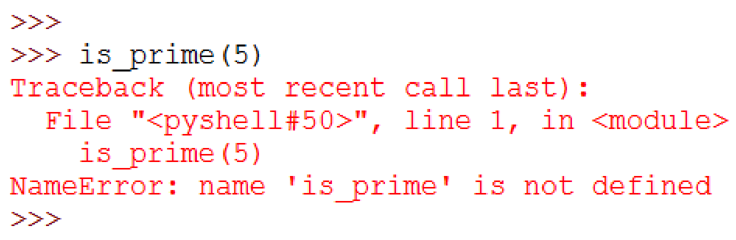
Here is the **<u>syntax</u>**.

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

code block

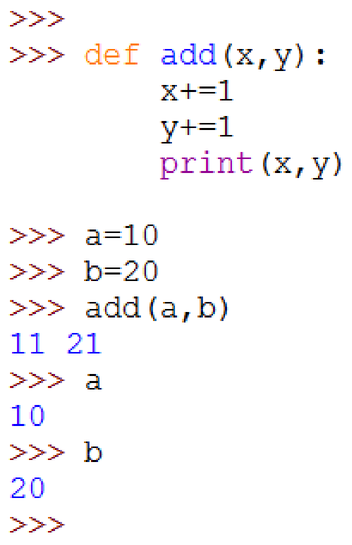
And the **<u>semantics</u>**:

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 "**<u>passed</u>**" 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 **<u>local</u>**. 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 subtler than **Notice1**. We have passed in simple types. Stay tunedfor what happens later on when we pass mutable objects.

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

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

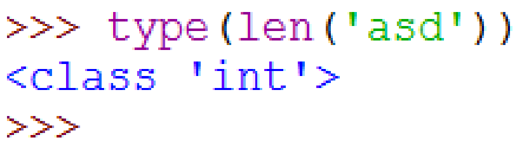
- **return <some value>**
- **return**
- **or … the function just "falls off" the last statement and implicitly returns to the caller.**

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.

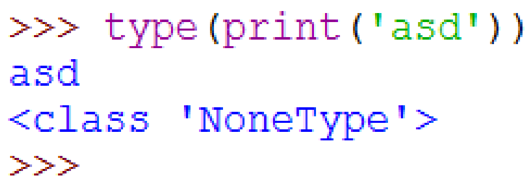
The third "form" behaves just like the one above.

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.

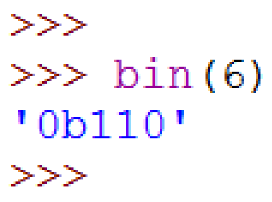
Answer:

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 string representation of n.But Leave out the leading ‘0b’ returned by the built in function bin.

Problem:

A generalization of the above.

Write a function

convert(a , basea, b)

where a is an integer in base a to an equivalent integer in base b. The result will be a base b interger represented as a string. basea and baseb are integers between 2 – 9.

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.

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. Unless you are in a function

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.

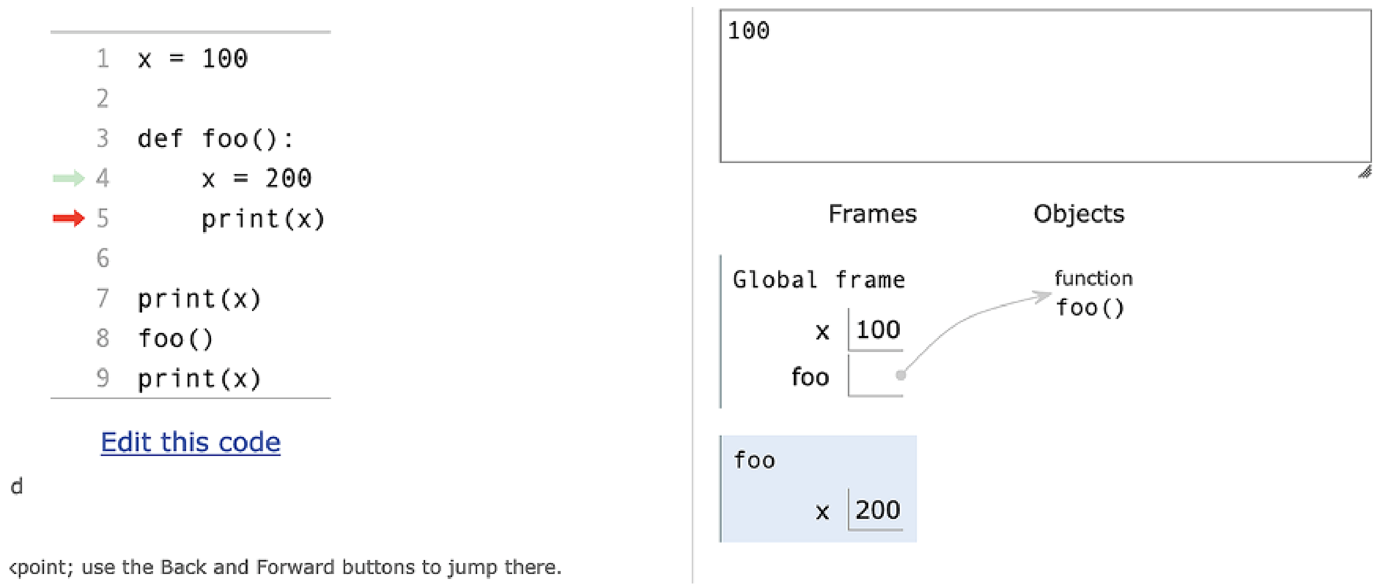
LOCAL VARIABLES

Firstly, function parameters have local scope, so that any changes to them do not affect their “original” value outside the function. This is like 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.



```
x = 100

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

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

```
100
```

THE GLOBAL STATEMENT

<u>What if, from within the function, I want to change the global variable?</u>

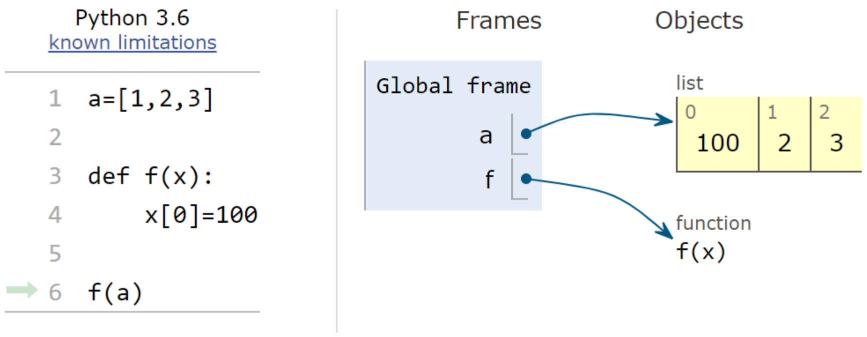
Use the global statement.

```
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.



```
a=[1,2,3]

def f(x):
    x[0]=100

f(a)
```

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? <u>This inner function and its accessible variables etc. sometimes known as a closure</u>, is a function that's defined when foo is <u>executed</u>. 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 **<u>LEGB:</u>**

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". There is **no need** to declare a variable nonlocal if it is **only going to be read**.

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.

What is the difference between a nonocal reference of an inner function and a closure?

When you define an inner function within an outer function, the inner function can access variables from the outer function's scope. These accessed variables are known as "nonlocal" to the inner function because they are not defined within the inner function's local scope nor are they global. They lie in the enclosing function's scope.

Nonlocal References in Inner Functions

Here's a simple example:

```
def outer():
    x = "outer x"
    def inner():
        print(x)  # Accessing the nonlocal variable x from the outer function

    inner()
outer()
```

When we run this we get: RUN IT!

Closures

A closure occurs when an inner function remembers and has access to variables from its enclosing scope even after the outer function has finished executing. The key point is that the inner function has "closed over" the variables from its enclosing scope.
This "closing over" behavior enables the inner function to retain the state of its environment even when called outside its original context. For a function to be a closure, it must satisfy three conditions:

1. It must be a nested function.
2. It must access variables from an enclosing scope.
3. The inner function must be returned or passed out of its enclosing function.

Here's an example that demonstrates a closure (without the nonlocal we used above):

```
def outer_function(x):
    y = 5

    def inner_function(z):
        return x + y + z  # Accesses x and y from the outer scope

    return inner_function  # Returns inner_function itself, not a value

# Creation of a closure
closure = outer_function(10)
```

In some more detail …

Code:

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

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

What is going on?

1. Definition of `foo`:

- `foo(x)` is a function that takes a single argument `x`.
- Inside `foo`, another function `bar(y)` is defined.
    - `bar(y)` takes another argument `y` and returns the product of `x` and `y`.
- `foo(x)` returns the function `bar`.

2. Calling `foo`:

- When `foo(10)` is called, `x` is set to `10`.
- The function `bar(y)` is returned by `foo`, and it **remembers the value of `x` (which is `10`)** even after `foo` finishes execution.
    - This is called a **closure**: `bar` retains access to the variable `x` from the surrounding scope of `foo`.

3. Assigning `bar` to `f`:

- `f = foo(10)` assigns the **returned `bar` function** to the variable `f`.
- Now, `f` is a function that takes one argument `y` and computes `10 * y`.

4. Calling `f(20)`:

- When `f(20)` is called, it executes the `bar(y)` function:
    - The `x` value is remembered as `10` (from the closure).
    - The `y` value is `20`.
    - So, `x * y` becomes `10 * 20`, which equals `200`.

5. Printing the Result:

- `print(f(20))` prints the result of `f(20)`, which is `200`.

Note:

1. **Closure**: The inner function `bar` retains access to the variable `x` from the outer function `foo` even after `foo` has finished execution.
2. **Function as a First-Class Object**: The `bar` function is returned as a value and assigned to `f`, allowing it to be called independently.

Final Output:

```
200
```

1. The Closure Mechanism

When a nested function (like `bar`) uses a variable from its enclosing scope (like `x` in `foo`), Python creates a **closure** for the nested function. This closure stores references to the variables from the enclosing scope that the nested function needs.

2. Where is `x` Stored?

- The variable `x` is not stored in the global or local scope of `bar`. Instead:
    - It is captured and stored in a **cell object** within the `__closure__` attribute of the function `bar`.
    - This ensures that `x` remains accessible even after the enclosing function `foo` has finished executing.

3. Inspecting the Closure

You can see this in action by inspecting the `__closure__` attribute of `f`:

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

f = foo(10)  # Call foo and get the inner function

# Inspect the closure
print(f.__closure__)  # Output: (<cell at 0x...: int object at 0x...>,)
print(f.__closure__[0].cell_contents)  # Output: 10
```

- `f.__closure__` is a tuple of **cell objects**, where each cell represents a variable captured from the enclosing scope.
- Each `cell` contains a reference to the value of the variable (in this case, `x = 10`).

4. Data Structure: The Cell Object

- The **cell object** acts as a container for the variable `x`.
- Internally, Python uses this cell object to maintain a reference to the captured variable, ensuring it is accessible for the nested function.

5. Key Points About This Storage:

1. **Efficient Memory Management**: Python stores only a reference to `x`, not a copy, allowing efficient use of memory.
2. **Garbage Collection**: The captured variable is not garbage collected as long as the closure exists, even if the enclosing function (`foo`) has finished executing.

Functions are objects and …

Can have attributes assigned to them. For example:

```
def do_thing():
  return
do_thing.whatever = "hi"

print(do_thing.whatever)
#> hi
```

Note that not all objects can have attributes assigned to them.

Python explicitly forbids attribute assignment to **built-in functions**:

print.some_data = "foo"

> AttributeError: 'builtin_function_or_method' object has no attribute 'some_data'

And you can’t assign attributes to built-in types:

var = "stringy string"

var.some_data = "foo"

> **AttributeError: 'str' object has no attribute 'some_data'**

Even though both are objects:

isinstance(print, object)

#> True

isinstance(var, object)

#> True

How does this work?

Internally, it's just a dictionary that handles failed attribute lookups (i.e., nondefault attributes). You can access or even replace such dictionary using **__dict__** attribute.

For example, you can track the number of times a function was called:

```
def func(a, b):
    func.ncalls += 1
    return a + b


func.ncalls = 0

func(1, 2)
func(3, 2)
print(func.ncalls)
```

Notice that ncalls is not a local variable in func, rather it is an attribute of the function object func.

Problem: Consider the following function

```
def Dog():
  def bark():
    print("bark bark")
  Dog.bark = bark
  return Dog
```

What happens when we run bark()?

What happens when we run Dog.bark()?

What happens when we run:

```
spot= Dog()
spot.bark()
```

What if we had **just** had:

```
def Dog():
  def bark():
    print("bark bark")
  Dog.bark = bark
  return Dog

Dog()
Dog.bark()
```

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.

Try this:

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

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.

Why those rules?

Say we define a function with a non-default parameter following a default parameter. This could lead to ambiguity in function calls.

```
# This function has invalid Python syntax
def my_func(a=10, b):
   print(f"a: {a}, b: {b}")
```

In this hypothetical situation, when calling **my_func** with a single argument, it would be unclear whether the provided argument should be assigned to **a** or **b**:

```
my_func(5)
```

- Should **5** be assigned to **a**, making use of the default value for **b** (which doesn't exist)?
- Or should **5** be assigned to **b**, using the default value for **a**?

This ambiguity arises because there's no clear rule to determine how to map the provided arguments to parameters when the parameters are not consistently defined (i.e., a mix of default and non-default parameters without clear ordering).

To avoid such ambiguities, Python enforces the rule that once you start defining function parameters with default values, all following parameters must either have default values as well or be part of the variable arguments (***args** or ****kwargs**). This rule ensures that there is always a clear and unambiguous mapping from provided arguments to the function's parameters, preserving the clarity and predictability of function calls.

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

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

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

<u>Notice how the default argument retains the modifications made from previous invocations</u> 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.

We can then write: factorial.__doc__, or help(factorial)

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:

```
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): # enumerate creates an iterator that yields pairs.
        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, not changing the passed list.
    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 just 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.

Argument unpacking when passing arguments.

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.

We will return to this topic in greater detail later on.

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.

Argument unpacking when returning values.
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

In some cases, when you return multiple values from a function, you might use a regular tuple. For example:

```
return (parts[0].strip(), parts[1].strip())
```

In this case, `parts[0]` refers to the first part of the string (the "name") and `parts[1]` refers to the second part (the "value"). However, this approach can be unclear when reading the code because you have to remember what `parts[0]` and `parts[1]` represent.

Wouldn't it be better if we could write something more readable, like this?

```
return (name.strip(), value.strip())
```

This makes it much clearer what each of parts[0] And parts[1] represents.

Well, we can’t to this exactly, but we can make it such that we can say name and value in the calling scope. How?

Enter Named Tuples

Named tuples give us the clarity of assigning names to the values we're returning, while maintaining the tuple-like behavior. Here’s how you can define a named tuple:

```
from typing import NamedTuple
```

Define a named tuple to hold the parsed result

```
class ParseResult(NamedTuple): # inherit from NamedTuple
    name: str # notice the optional type hint
    value: str
```

Now, instead of returning a regular tuple, we can return an instance of `ParseResult`:

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

Here, `ParseResult(parts[0].strip(), parts[1].strip())` creates a `ParseResult` object with two fields: `name` and `value`. This makes the return value much clearer and more structured than using a basic tuple.

Accessing Named Tuple Fields

A named tuple works just like a normal tuple (you can still unpack it, iterate over it, etc.), but you can also reference its fields using meaningful names:

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

This prints:

```
url http://www.python.org
```

By using `r.name` and `r.value`, it's immediately clear what each value represents. This eliminates any confusion that might arise from using index-based access like `r[0]` or `r[1]`.

Named tuples provide a simple, readable, and structured way to return multiple values from a function. They enhance code clarity by allowing you to refer to values by name rather than relying on index positions, making your code easier to maintain and understand. You can still refer to the various elements of the tuple by positional value if you like.

Function definitions

What does Python do when it encounters a function definition?

When the Python interpreter encounters a function definition in a program for the first time, a series of steps are followed to interpret and store the function for later use. Here's a detailed breakdown of what happens:

1. Parsing the Function Definition

The interpreter first parses the function definition. This involves analyzing the syntax of the `def` statement, including the function name, parameters, and the body of the function. Python's parser converts this source code into an abstract syntax tree (AST), which represents the structure of the code in a tree-like form.

2. Compilation to Bytecode

After parsing, the function's code block (its body) is compiled into bytecode. Bytecode is a low-level, platform-independent representation of the source code, which is designed to be executed by the Python Virtual Machine (PVM). Each operation in the function body, such as variable assignment, operation on variables, and function calls, is translated into a series of bytecode instructions.

3. Creation of a Function Object

Once the bytecode is ready, Python creates a function object. This function object is a first-class object, meaning it can be passed around and manipulated like any other object in Python. The function object contains several pieces of information:

What's in the function object?

Code object: This contains the compiled bytecode of the function, as well as other metadata such as the function's name, its argument names, and its defaults.
Global references: The function object also keeps a reference to the globals of the module in which it is defined. This is important because the function will need access to global variables and other functions defined at the module level when it is called.
Closure: If the function is a closure, the function object will also contain a reference to any variables captured from an enclosing scope.

4. Function Object Assignment

The function object is then bound to the function's name in the current namespace. This means that after the definition is interpreted, you can call the function by using its name. In Python, namespaces are implemented as dictionaries, so the function name is a key in this dictionary, and the function object is the value.

5. Ready for Execution

At this point, the function is fully defined and ready to be executed. However, it's important to note that the function's code has not been executed yet; the function will only be executed when it is called.

When you call the function, the interpreter creates a new execution frame for that function call, pushing it onto the call stack. This frame contains its own namespace for local variables, references to any global or nonlocal variables it needs, and a pointer back to the function object's code so that the interpreter knows what bytecode to execute.

In summary, when the Python interpreter encounters a function definition for the first time, it parses the definition, compiles the body of the function into bytecode, creates a function object containing this bytecode and other relevant information, and then binds this object to the function's name in the current namespace. This process makes the function ready for execution whenever it is called later in the program.

Now … above we have **1. Parsing the Function Definition. What does that look like?**

First, Python creates an Abstract Syntax Tree (AST) representation of the abstract syntactic structure of the source code Each node in the tree denotes a construct occurring in the source code.

For example, consider:

```
def add(a, b):
    return a + b
```

The AST for this function would represent the structure of the function in a hierarchical manner, breaking down the function definition, parameters, body, and return statement. A simplified version might look something like this:

```
FunctionDef
    ├── name: "add"
    ├── arguments
    │    ├── arg: "a"
    │    └── arg: "b"
    └── body
         └── Return
              └── BinOp
                   ├── Left: Name(id='a', ctx=Load())
                   ├── Op: Add()
                   └── Right: Name(id='b', ctx=Load())
```

In this tree:

. The **root** is a `FunctionDef` node, representing the function definition.
. The `**FunctionDef**` node has children representing the function's name (`add`), its arguments (`a` and `b`), and its body.
. The `**arguments**` node lists all parameters the function accepts. In this case, there are two arguments, `a` and `b`.
. The `**body**` node contains a `Return` node, indicating that the function will return a value.
. The `**Return**` node has a child `BinOp` (binary operation) node, representing the addition operation.
. The `**BinOp**` node has three children: `Left`, `Op`, and `Right`. `Left` and `Right` are `Name` nodes representing the operands (the parameters `a` and `b`), and `Op` is an `Add` node representing the addition operator.

In the context of the Abstract Syntax Tree (AST) for Python, `ctx` stands for "context", and it indicates how a variable or name is being used in that particular part of the code. The context can be one of several types, indicating whether a variable is being read from, assigned to, or deleted, among other possibilities.

In the AST snippet above, the `ctx` is specifically marked as `Load()`, which means the variables `a` and `b` are being loaded from their respective locations (e.g., memory, a scope, etc.) for use in an expression. In this case, they are being used as operands in a binary addition operation.

The different types of contexts in Python's AST include: Load, Store, Del, AugLoad, AugStore, and Param.

Python will show you a user-friendly representation of the AST by using the **ast** module like this:

```
import ast
source_code = '''
def add(a,b):
    return a+b
'''
tree = ast.parse(source_code)
```

Python will produce this:

```
Module(
    body=[
        FunctionDef(
            name='add',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(arg='a'),
                    arg(arg='b')],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Return(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Name(id='b', ctx=Load())))],
            decorator_list=[])],
    type_ignores=[])
```

This structure makes it clear how the various parts of the function are organized and related to each other, which is useful for the interpreter during the parsing and compilation stages.

Then we have 2. Compilation to Bytecode. What does that look like?

The bytecode generated for the `add` function is as follows:

1. `LOAD_FAST` (a): This instruction loads the local variable `a` onto the stack. It's the first argument of the function `add`.
2. `LOAD_FAST` (b): This instruction loads the next local variable `b` onto the stack. It's the second argument of the function `add`.
3. `BINARY_ADD`: This instruction pops the top two values from the stack (which are `a` and `b`), adds them, and then pushes the result back onto the stack.
4. `RETURN_VALUE`: This instruction returns the value on top of the stack to the caller of the function.

Each `Instruction` in the bytecode includes the operation name (`opname`), the operation code (`opcode`), the argument to the operation if any (`arg` and `argval`), a representation of the argument (`argrepr`), the offset in the bytecode, and whether the line starts a new line of Python code or is a jump target.

This sequence of instructions is what the Python interpreter executes when the `add` function is called, performing the addition operation and returning the result.

The bytecode is essentially a tuple consisting of the following components:

1. Operation Name (`opname`)

The operation name (`opname`) is a human-readable string representing the operation that the bytecode instruction performs. For example, `LOAD_FAST` is the operation name for the opcode that loads a local variable onto the stack. These names are meant to be descriptive and help in understanding the purpose of the opcode without needing to know its numerical code.

2. Operation Code (`opcode`)

The operation code (`opcode`) is a numerical value that represents the specific operation to be performed. Each type of operation (like loading a variable, performing an addition, or returning a value) has a unique opcode. The Python interpreter uses this code to identify what action to take when executing the bytecode. For instance, in CPython, `LOAD_FAST` might have an opcode value of 124, although the exact number can vary between Python versions.

3. Argument (`arg` and `argval`)

arg: This is the raw numerical argument for the opcode, if any. Some operations require additional information to be executed properly. For example, `LOAD_FAST` needs to know which local variable to load. The `arg` value provides this information, usually as an index or a reference to other data structures like the constant pool or variable names list.

argval: This represents the interpreted value of the argument. It provides a more meaningful representation of the `arg` value. For instance, if `arg` is an index of a local variable, `argval` would be the name of that variable.

4. Argument Representation (`argrepr`)

The argument representation (`argrepr`) is a human-readable description of the argument value (`argval`). It's designed to make the bytecode more understandable. For example, if the `argval` is the name of a local variable like `'x'`, then `argrepr` would also be `'x'`, making it clear that the operation involves the variable `'x'`.

5. Offset

The offset indicates the position of the bytecode instruction within the bytecode sequence. It's essentially the "address" of the instruction in the bytecode. This is useful for understanding the flow of execution, especially for operations that involve jumps, as it tells you where in the bytecode sequence an instruction is located.

6. Line Starts and Jump Targets

Line Starts: This indicates whether the instruction is the first one on a new line of source code. It's useful for debugging and profiling, as it helps map bytecode instructions back to the lines of source code they came from.

Jump Targets: This indicates whether the instruction is a target for jumps from other instructions. Control flow operations like loops and conditionals involve jumps in the bytecode. An instruction marked as a jump target is where the control flow might jump to during execution.

Aside: Load_FAST

`LOAD_FAST` is an opcode used in Python's bytecode instruction set, which is part of the Python Virtual Machine (PVM). The `LOAD_FAST` opcode is used to load a local variable onto the stack quickly. It's designed to be an efficient way to access local variables within a function.

Here's a breakdown of how `LOAD_FAST` works:

Local Variable Access: Each function in Python maintains its own local variables. These are stored in a fixed-size array where each local variable can be accessed by its index. The `LOAD_FAST` instruction takes advantage of this by using the variable's index to access it directly.
Stack: The Python Virtual Machine (PVM) uses a stack-based execution model. Operations are performed by pushing operands onto the stack and then executing an instruction that operates on those operands. The results of operations are then pushed back onto the stack.
Efficiency: The `LOAD_FAST` opcode is optimized for speed. Since local variables are frequently accessed during function execution, having a fast way to load these variables onto the stack is crucial for performance. By using the index of the variable in the local environment, `LOAD_FAST` can quickly fetch the variable's value and push it onto the stack.

In the context of the bytecode for the `add` function:

```
LOAD_FAST          a
LOAD_FAST          b
```

- The first `LOAD_FAST` instruction loads the value of the local variable `a` onto the stack.
- The second `LOAD_FAST` instruction loads the value of the local variable `b` onto the stack.

After these instructions, the top two elements on the stack are the values of `a` and `b`, ready to be used by subsequent operations, such as `BINARY_ADD` in the case of the `add` function. This opcode is a critical part of the Python bytecode execution model, contributing to the efficiency of function calls and variable access within functions.

An example – remember the pythonic swap”

```
def swap(a, b):
    a, b = b, a # tuple packing
    return a, b
```

Here's a breakdown of how this is implemented at the bytecode level:

1. **Tuple Packing**: First, the values of `b` and `a` are placed onto the stack, and then Python creates a tuple from these values. This is the packing step, where `b, a` becomes `(b, a)`.

2. **Tuple Unpacking**: Python then unpacks this tuple directly into the variables `a` and `b`. The first element of the tuple (originally `b`) is assigned to `a`, and the second element (originally `a`) is assigned to `b`.

To actually see what is going on we can disassemble this function, and see the bytecode instructions that Python uses to perform the swap:

First:

```
import dis
dis.dis(swap)
```

The disassembled bytecode might look something like this (exact output could vary slightly depending on the Python version):

```
4          0 LOAD_FAST               1 (b)
           2 LOAD_FAST               0 (a)
           4 ROT_TWO
           6 STORE_FAST              0 (a)
           8 STORE_FAST              1 (b)

 5        10 LOAD_FAST               0 (a)
          12 LOAD_FAST               1 (b)
          14 BUILD_TUPLE             2
          16 RETURN_VALUE
```

Here's what happens in these instructions:

- `LOAD_FAST` instructions push the values of `b` and then `a` onto the stack.
- `ROT_TWO` swaps the top two elements of the stack. After this instruction, the top of the stack has the original value of `a`, and the second element from the top has the original value of `b`.
- `STORE_FAST` instructions then pop these values off the stack and store them back in the local variables `a` and `b`, effectively swapping their values.

In detail:

Line 4
`0 LOAD_FAST 1 (b)`: This instruction loads the value of the local variable `b` (at index 1 in the local variables array) onto the top of the Python Virtual Machine (PVM) stack.

`2 LOAD_FAST 0 (a)`: Following that, the value of the local variable `a` (at index 0) is loaded onto the stack, above the previously loaded value of `b`.

`4 ROT_TWO`: This instruction rotates the top two stack items. After this operation, the value of `a` (which was loaded second) is now below the value of `b` on the stack, effectively swapping their positions in the stack without using a temporary variable.

`6 STORE_FAST 0 (a)`: The top value on the stack (which is now the original value of `b` due to the `ROT_TWO` operation) is popped off the stack and stored in the local variable `a` (at index 0).

`8 STORE_FAST 1 (b)`: Similarly, the next value on the stack (which is now the original value of `a`) is popped off and stored in the local variable `b` (at index 1). At this point, the values of `a` and `b` have been swapped.

Line 5
`10 LOAD_FAST 0 (a)`: The value of `a` is loaded onto the stack again. Note that `a` now holds the original value of `b` due to the swap.

`12 LOAD_FAST 1 (b)`: The value of `b` is loaded onto the stack, above the value of `a`. Remember, `b` now holds the original value of `a`.

`14 BUILD_TUPLE 2`: This instruction pops the top two values from the stack (the values of `a` and `b`) and builds a tuple from them. This tuple is then pushed onto the stack.

`16 RETURN_VALUE`: Finally, the top value on the stack (which is now the tuple containing the swapped values) is returned from the function.

Note:

The designation "Line 4" in the bytecode disassembly output corresponds to the **line number in the original Python source code from which the bytecode was generated**. It does not indicate the fourth line of the bytecode itself but rather that the bytecode instructions starting from that point correspond to the fourth line of the source Python code.

The numbers preceding the instructions in the bytecode disassembly, such as the 4 in 4 ROT_TWO, represent the **byte offset of each instruction within the bytecode sequence**. This offset indicates the position of the instruction in the bytecode and is used by the Python Virtual Machine (PVM) to navigate through the bytecode as it executes the program.

Here's a breakdown of what these numbers mean:

Byte Offset: The number is essentially an address within the bytecode sequence. It tells the PVM where each instruction starts. In the example, `4 ROT_TWO` means that the `ROT_TWO` instruction starts at byte offset 4 in the bytecode.

Sequential Execution: The Python interpreter reads and executes bytecode instructions sequentially, unless an instruction explicitly alters the flow of execution (e.g., through a loop or conditional jump). The byte offset helps the interpreter keep track of its position in the bytecode sequence.

Jump Targets: For control flow instructions that involve jumps (like `FOR_LOOP`, `JUMP_FORWARD`, or conditional jumps), the byte offset provides a target address to jump to. For example, if there's a jump instruction that says to jump forward by 6 bytes, the interpreter would move to the instruction 6 bytes ahead of the current instruction's byte offset.

Instruction Length: The difference between consecutive byte offsets also implicitly indicates the length of each bytecode instruction. Different instructions can have different lengths depending on whether they have arguments and how those arguments are encoded.

In the context of the above example, the `ROT_TWO` instruction at byte offset 4 follows the `LOAD_FAST` instruction at byte offset 2. The difference in offsets also tells you that the `LOAD_FAST` instruction (including any potential argument it might have) takes up 2 bytes.

Question: By looking at the bytecode it seems that the swap is being done without a third variable. How?

Function modification at runtime

Hold on to your seats ……

You can modify a function in real time in Python, (but this is generally done using higher-level constructs like decorators which modify the functions behavior while leaving the original function intact) by modifying the function's bytecode directly. Here is an example:

```
from bytecode import Instr, Bytecode
import types

def f(a, s):
   return a + s

# Convert the function's code to a mutable bytecode object
byte_code = Bytecode.from_code(f.__code__)

# Find the BINARY_ADD instruction and replace it with BINARY_MULTIPLY
for instr in byte_code:
    if isinstance(instr, Instr) and instr.name == "BINARY_ADD":
        instr.set("BINARY_MULTIPLY")
        break

f = byte_code.to_code()
f = types.FunctionType(f, globals(), "new_f")

# Test the modified function
print(f(2, 3))  # Should print '6' instead of '5'
```

How does this work?

1. `from bytecode import Instr, Bytecode`
   - This line imports the `Instr` and `Bytecode` classes from the `bytecode` library, which allows for manipulation of Python bytecode.

2. `import types`
   - This imports the `types` module, which provides utility functions and types for working with different Python object types, including functions.

3. `def f(a, s): return a + s`
   - Here is our sample function that we want to modify. `f` is defined that takes two arguments, `a` and `s`, and returns their sum.

4. `byte_code = Bytecode.from_code(f.__code__)`
   - This converts the code object of the function `f` (accessible via `f.__code__`) into a `Bytecode` object from the `bytecode` library. The `Bytecode` object is mutable, allowing for modifications to the bytecode.

5. The `for` loop iterates over each instruction in the `byte_code` object:
   - `for instr in byte_code:` iterates through each bytecode instruction in `byte_code`.

6. `if isinstance(instr, Instr) and instr.name == "BINARY_ADD":`
- This checks if the current instruction (`instr`) is an instance of the `Instr` class (indicating it's an instruction rather than a label or other bytecode component) and if the name of the instruction is `"BINARY_ADD"`, which represents the addition operation in Python bytecode.

7. `instr.set("BINARY_MULTIPLY")`
- If the condition in the previous line is true, this line changes the instruction from addition to multiplication by setting the instruction's name to `"BINARY_MULTIPLY"`. This effectively changes the operation performed by that instruction in the bytecode.

8. `break`
- This exits the loop after modifying the first `BINARY_ADD` instruction found, ensuring that only the first addition operation in the function is changed to multiplication.

9. `f = byte_code.to_code()`
- Converts the modified `Bytecode` object back into a code object suitable for execution by the Python interpreter.

10. `f = types.FunctionType(f, globals(), "new_f")`
- This creates a new function from the modified code object. `types.FunctionType` constructs a new function object, `f`, using the provided code object, the current global namespace (`globals()`), and the name `"new_f"` for the new function.

11. `print(f(2, 3))  # Should print '6' instead of '5'`
- Finally, the modified function `f` is called with arguments `2` and `3`. Since the addition operation in the original function `f` has been changed to multiplication, the expected output is `6` (the product of `2` and `3`), instead of `5` (the sum of `2` and `3`).

Creating and running code at runtime – exec and eval

There is a mechanism in python to read in a string, representing a function, and at runtime, and then call that function by name.

`exec()` and `eval()` are both built-in functions in Python that allow for the dynamic execution of Python code, but they serve different purposes and have distinct behaviors:

exec()

- `exec()` is used to execute dynamically generated Python code which can be a single statement, a statement block, or even a string representing a Python script.
- It does not return any value; it only executes the code within its argument.
- It can be used for executing dynamic Python code that includes loops, conditionals, function/class definitions, and so forth.
- It can modify the current scope if used without specifying an explicit namespace, meaning it can define new variables or functions or change existing ones within the scope it is called.

Syntax example: `exec("code")`, where `"code"` is a string containing Python statements.

eval()

- `eval()` is used to evaluate valid Python expressions (not statements) contained in a string and return the result of the expression.
- It is essentially used for simple expression evaluation, like arithmetic calculations or evaluating expressions to a single value.
- It cannot execute complex Python code like loops, conditionals, function/class definitions, etc.
- It's useful for dynamically evaluating expressions that result in a single value, such as `'3 + 4'` or even calling functions that return a value.

Syntax example: `result = eval("expression")`, where `"expression"` is a string containing a Python expression, and `result` will store the evaluated result.

Key Differences

Usage: `exec()` is for executing statements (including multi-line code blocks, function definitions, etc.), whereas `eval()` is for evaluating expressions and returning their results.
Return Value: `exec()` does not return any value (or returns `None`), while `eval()` returns the value of the given expression.
Scope Modification: `exec()` can modify the current scope or namespace, while `eval()` is generally used for expressions and has limited capability to modify the current scope.

Careful: Due to their ability to execute arbitrary code, both `exec()` and `eval()` should be used with caution, especially with untrusted input, to avoid potential security vulnerabilities such as code injection attacks.

exec() example:

```
# Define a string representing a block of Python code
code_str = """
def calculate_area(length, width):
    return length * width

def print_greeting(name):
    if name:
        greeting = f"Hello, {name}!"
    else:
        greeting = "Hello, stranger!"
    print(greeting)
"""

# Dynamically compile and execute the code block
exec(code_str)

# Now, the functions 'calculate_area' and 'print_greeting' are defined and can be called
area = calculate_area(10, 5)  # Calling the dynamically defined function
print(f"Area: {area}")

print_greeting("Alice")  # Calling another dynamically defined function
```

eval() example:

```
# Define an arithmetic expression as a string
expression = "(3 + 5) * 2 / (4 - 2)"

# Use eval() to evaluate the expression
result = eval(expression)

print(f"The result of the expression {expression} is: {result}")
```

Scripts and Modules

In languages like C++ or Java a distinction is made between **application programs** and **libraries**.

For example, in C++ you would write a **main function** and perhaps some supporting functions to accomplish a particular task. This is an application. The Python equivalent is a **script**.

If you needed some specific functionality, say input/output, you would **#include<iostream>**. iostream is not a "main program", an "application", but rather a "library", a file containing classes, functions, definitions, variable declarations and initializations. Its not meant to be run as a standalone application, but as a collection of reated functions to support a particular functionality. iostream supports stream-oriented i/o in C++. The Python equivalent is called a **module**. The math module is an example.

Scripts are run as top-level applications, modules are imported. Both are .py files, the difference is in their intended use.

You might have seen the following in python programs:

```
if __name__ == "__main__"
```

What is this and how does this relate scripts and modules.

Do this: Make two files, module.py and import_module.py.

```
# module.py
def foo():
   print("Function foo from module.py")

if __name__ == "__main__":
   # This block of code will run only when module.py is executed directly,
   # not when it is imported in another file.
   foo()
   print("module.py is being run directly")
```

```
# import_module.py
import module
module.foo()
```

1. When `module.py` is run directly (at the "top level")we get:

```
Function foo from module.py
module.py is being run directly
```

2. But if we run **import_module**

we get:

```
Function foo from module.py
```

What is going on?

The `if __name__ == "__main__":` statement in Python serves a crucial purpose when it comes to writing Python modules that can be run as **scripts** or **imported** into other **modules**.

Script vs. Module: Python files can act both as reusable modules and as standalone scripts. `if __name__ == "__main__":` allows a Python file to distinguish between these two use cases, executing some code only when the file is run as a script and not when it's imported as a module.

Encapsulation: This statement encapsulates the "executable" part of the code, making it clear which part of the module is meant to execute as a script. This is especially useful for readability and maintainability.

How It Works

`__name__` Variable: When a Python file is executed, Python sets several special variables, and `__name__` is one of them. If the file is being run as a script, `__name__` is set to `"__main__"`. If the file is being imported as a module into another file, `__name__` is set to the name of the file/module.

Use Cases

1. **Running Tests**: You can include a test suite within the `if __name__ == "__main__":` block of a module, allowing you to run tests only when the module is executed directly.

2. **Demonstration and Documentation**: For modules that define functions, classes, or other components, you can include examples of how to use these components within the `if __name__ == "__main__":` block. This serves both as a demonstration of functionality and as basic usage documentation.

3. **Utility Scripts:** When creating utility scripts that can also be imported as modules, use this block to contain the script logic, allowing the script to be both directly executable and importable for use in other modules.

4. **Application Entry Point**: In larger applications, the `if __name__ == "__main__":` block can be used in the main application file as the entry point to the application, invoking the main function or starting the main application loop.

Sometimes this idiom might look like this:

```
# Define a function to do something
def greet(name):
    print(f"Hello, {name}!")

# define some more functions to do other things

……. And perhaps more functions

# Define the main function to encapsulate the script's primary logic

def main():
    # Call the greet function with a name
    greet("Alice")
  # You can add more logic here in the main that should execute when the script runs directly

# This idiom checks if the script is being run directly (not imported as a module)
if __name__ == "__main__":
    # If so, call the main function
    main()
```

This structure is similar to the `main()` function in C++ programs, which serves as the **entry point for execution**.

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 of 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 s to 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 x in s.
s.index(x [,start [,stop]])	Returns the smallest i where s[i]==x. start and stop optionally specify the starting and ending index for the search.
s.insert(i,x)	Inserts x at index i.
s.pop([i])	Returns the element i and removes it from the list. If i is omitted, the last element is returned.
s.remove(x)	Searches for x and removes it from s.
s.reverse()	Reverses items of s in place.
s.sort([keyf [, reverse]]])	Sorts items of s in place. Keyf is a key function. Reverse is a flag that sorts thelist 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??

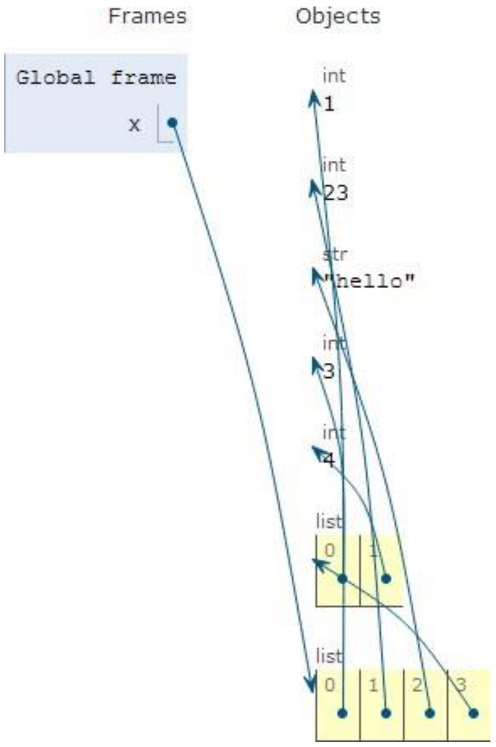
Example:

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 "[" ,"]".

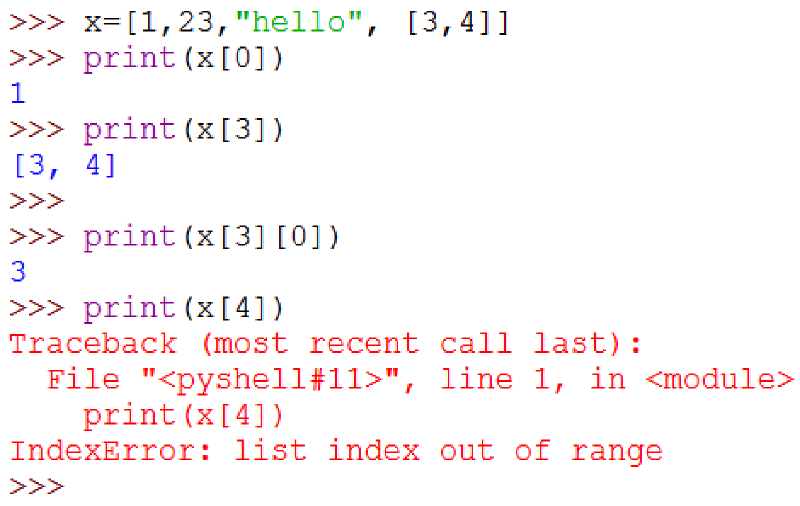
Question:

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

Answer:

We use **<u>square brackets with an integer</u>** 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.

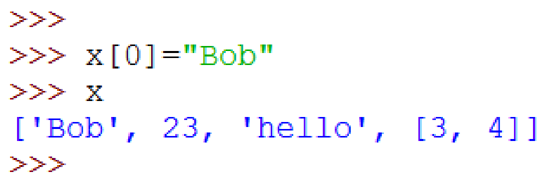
3. Since there is no element in the list x[4] (they are x[0], x[1], x[2], x[3]) we are trying to access a nonexistent element and Python prints an error message.

Question:

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

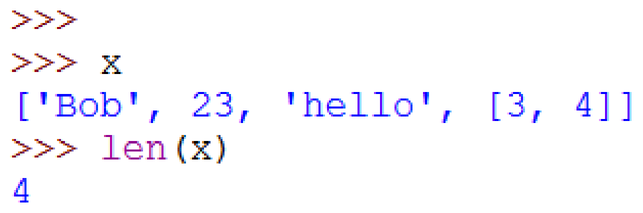
Answer:

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 list by using the len() function:



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

We say that a list is **<u>mutable</u>**. This means that is can be modified (i.e. “mutated).

Problem:

Given:

a=[[1,2],[3,4],[5,6]]

what is :

list(a)

list(list(a))

list((list(a)))

Why?

What about:

[a]

[[a]]

[[[a]]]

Why?

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

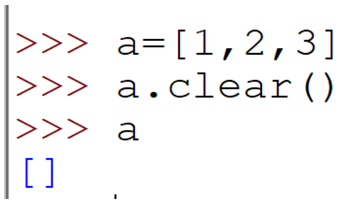
```
>>>
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.append("hello")
>>> a
[1, 2, 3, 4, 'hello']
>>> b=[5,6,7]
>>> a.append(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7]]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7]
>>> a.extend(8)
Traceback (most recent call last):
  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!

del and .clear

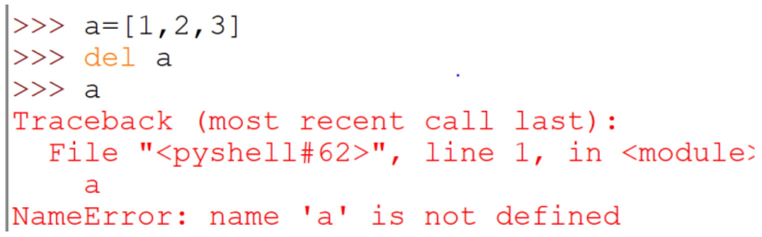
Say we have a=[1,2,3]

If we write **a.clear()** then a will be [].



```
>>> a=[1,2,3]
>>> a.clear()
>>> a
[]
```

But if we write **del a** we get:

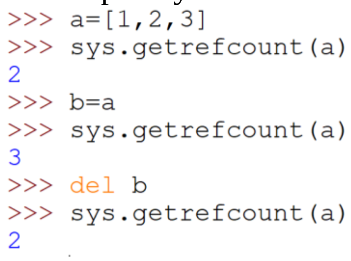


```
>>> a=[1,2,3]
>>> del a
>>> a
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module
    a
NameError: name 'a' is not defined
```

Remember in Python everything is an object and names are really pointers to objects. And an object may be pointed to many times.

Each object has a **reference count** and you can check the reference count of an object using the **getrefcount()** function from the **sys** module.

```
>>> import sys
```



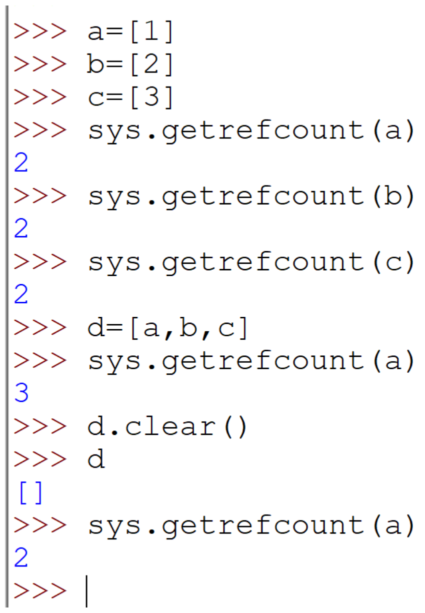
```
>>> a=[1,2,3]
>>> sys.getrefcount(a)
2
>>> b=a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

This function returns the reference count of the object passed to it. It's important to note that **getrefcount()** itself adds a temporary reference to the object, so the count returned will be one higher than the number of references you might expect.

Notice that del b didn't delete the list. When you use the del statement in Python, such as with del i, it unbinds the name i from the object it references. Essentially, del removes the binding of a name from the local or global namespace, depending on where the name was defined. It does not directly delete the object itself; rather, it just removes one reference to the object.

When the reference count of an object goes to zero, its marked eligible for garbage collection.

What about .clear()?



```
>>> a=[1]
>>> b=[2]
>>> c=[3]
>>> sys.getrefcount(a)
2
>>> sys.getrefcount(b)
2
>>> sys.getrefcount(c)
2
>>> d=[a,b,c]
>>> sys.getrefcount(a)
3
>>> d.clear()
>>> d
[]
>>> sys.getrefcount(a)
2
>>> |
```

Problem:

a, b, and c are as above. What will happen if we run the following:

```
d=[a,b,c]
for i in d:
   del i

print(d)
```

Why?

So, to summarize:

Note: In this summary I include data types that we haven't studied in detail yet, however we can refer to this later on.

The **del** statement and the **.clear()** method serve different purposes in Python and operate on different types of targets:

del Statement

- **Purpose**: The **del** statement is used to delete objects in Python. It can be used to remove individual items from a list or to delete entire variables or objects from the namespace.
- **Applicability**:
    - **Variables**: You can use **del** to remove a variable and its reference to an object from the namespace, which can potentially free up the object for garbage collection if there are no other references to it. For example, **del a** removes the variable **a**.
    - **List Items**: You can delete an item from a list by its index, e.g., **del my_list[2]**, which removes the item at index 2 from **my_list**.
    - **Slices**: You can remove a slice from a list, e.g., **del my_list[1:3]**, which removes items from index 1 up to but not including index 3.
    - **Dictionary Entries**: You can delete a key-value pair from a dictionary, e.g., **del my_dict['key']**, which removes the specified key and its associated value from **my_dict**.
    -

.clear() Method

- **Purpose**: The **.clear()** method is used to remove all items from **a mutable collection**, leaving the collection empty but still defined in the namespace.
- **Applicability**:
    - **Lists**: When you call **.clear()** on a list, it removes all elements, making the list empty (**[]**). For example, **my_list.clear()** clears all elements from **my_list**.
    - **Dictionaries**: Calling **.clear()** on a dictionary removes all key-value pairs, leaving an empty dictionary (**{}**). For example, **my_dict.clear()** clears all entries from **my_dict**.
    - **Sets**: Similar to lists and dictionaries, calling **.clear()** on a set removes all elements, leaving an empty set (**set()**).

Key Differences

- **Scope**: **del** can be used more broadly to delete variables or specific items within objects, whereas **.clear()** specifically empties an entire mutable collection.
- **Effect on Namespace**: **del** can remove the name binding from the namespace entirely (e.g., deleting a variable), while **.clear()** only affects the contents of the object but leaves the variable in the namespace, now referring to an empty collection.
- **Type of Operation**: **del** is a statement in Python, while **.clear()** is a method that is called on objects of specific types (like lists, dictionaries, and sets).

The takeaway: use **del** when you need to remove items or variables, and use **.clear()** when you want to empty a mutable collection but keep the empty collection around.

A peek under the hood …

What happens internally when we write something like this:

```
d=[1,2,3]
del d[0]
```

When you perform the operation `del d[0]` on a list `d` in Python, several internal steps are involved to update the list:

1. **Removal of the Element**: The first element in the list (`d[0]`) is removed. This operation decreases the reference count of the object that `d[0]` was pointing to. If this was the only reference to that object, and no other references exist elsewhere in your program, the object becomes eligible for garbage collection.

2. **Shifting of Subsequent Elements**: To maintain the contiguous nature of the list, all elements that followed the removed element (`d[1]`, `d[2]`, etc.) are shifted one position to the left. This means that the element that was at `d[1]` moves to `d[0]`, `d[2]` moves to `d[1]`, and so on. This shift ensures there are no "gaps" in the list.

3. **Size Adjustment**: Internally, the list object adjusts its recorded size to reflect the removal of an element. The capacity of the underlying array that stores the list elements (which is typically larger than the number of elements to accommodate growth) might not change immediately upon the deletion of a single element. The Python list implementation may keep some empty space in the underlying array for efficient addition of new elements.

It's worth noting that the actual memory allocation and deallocation are handled by the Python memory management system and can be influenced by various factors, including implementation details of the Python interpreter you're using (like CPython, PyPy, etc.). The Python list is implemented as a dynamic array, which means it can grow or shrink as elements are added or removed, but the resizing of the underlying array (either growing or shrinking its allocated memory) doesn't necessarily happen with every addition or removal operation due to considerations for efficiency and performance.

To summarize: `del d[0]` removes the first element from the list `d`, shifts subsequent elements to fill the gap, adjusts the size of the list, and may eventually lead to adjustments in the allocated memory for the list, depending on the implementation and the state of the list.

Problem:

Here is an attempt to write my own version of clear(). Does my version accomplish the same thing as the built in one? Why or why not. (what about sub-lists …)

```
def my_clear(lst):
    for i in range(len(lst) - 1, -1, -1):
        del lst[i]

a = [1, 2]
b = [3, 4]
c = [a, 5, b, 6]

# Using my_clear
my_clear(c)
```

Interned objects

As noted above, when the reference count of an object goes to zero, its marked eligible for garbage collection. **There is an interesting exception**: the “**interned**” values. Consider:

```
i=5
del i
```

i now has a reference count decreased by 1. If there are no other references to the integer 5 elsewhere in your program, its reference count would become 0, making it eligible for garbage collection. However, in Python, small integers (typically between -5 and 256) are **interned**, meaning they are reused and never garbage collected, so the reference count change for the integer 5 will not lead to its deallocation.

In Python, small integers (typically in the range of -5 to 256) are implemented as "interned" objects. This means that instead of creating a new object each time a small integer is needed, Python uses a pre-existing, pre-allocated object for that integer value. These interned integers are, in a sense, singleton objects for their respective values, and any reference to such an integer actually points to the same object in memory.

Interning Details

1. Interned objects are singletons: When you assign a small integer to a variable, Python doesn't create a new integer object; it assigns a reference to the existing interned object representing that integer. This is efficient in terms of both memory usage and execution speed, as it reduces the number of integer objects that need to be created and managed.

2. Attributes: Even though integers in Python are objects and have attributes and methods (you can use `dir(1)` to see them), the way Python handles small integers does not affect how you interact with them. You can use these integers just like any other number in Python, and the fact that they are interned is mostly transparent to the user.

3. Pointers: Variables assigned with small integers essentially hold pointers to these pre-allocated integer objects. When you do `a = 1`, `a` holds a pointer to the interned object for `1`.

4. Universally Available: These interned integer objects are created at the start of the Python interpreter and are available globally. Any reference to a small integer in any part of a Python program will point to the same interned object.

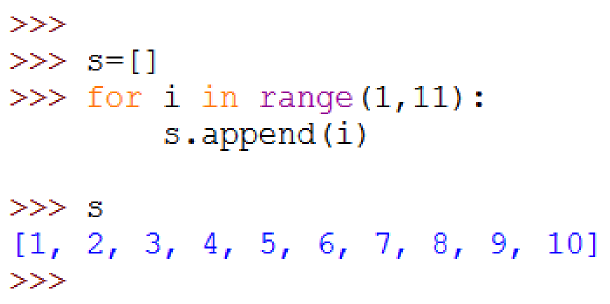
5. Comparison and Identity: Because all references to a given small integer point to the same object, both equality (`==`) and identity (`is`) checks work the same for these interned integers. For example, `(1 == 1)` and `(1 is 1)` both evaluate to `True`, not just because the values are equal, but also because both expressions are referring to the same object in memory.

Beyond Small Integers

For integers outside of the small integer range, Python may create new integer objects each time. However, the behavior can be implementation-specific and may vary. In such cases, two variables with the same integer value might not necessarily point to the same object, and an identity check (`is`) could evaluate to `False`.

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.

Problem:

Write a function **delete_at_even_position(x)** where x is a list of integers. So that if when we run

d = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

delete_at_even_position(d)
print(d)

we get [1,3,5,7,9]

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 1.

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 old! It's named after the Greek mathematician Eratosthenes. The beauty of this algorithm lies in its simplicity and efficiency, making it one of the most efficient ways to find all primes smaller than 10 million or so, especially when the range is not too large.

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

as well as an animation of the algorithm.

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

The algorithm works by iteratively marking the multiples of each prime number starting from 2. The key insight is that every non-prime number is a multiple of some prime number. By systematically marking the multiples of each prime as non-prime, we can sift through a list of integers and identify the primes.

Here's a step-by-step explanation of how the algorithm works:

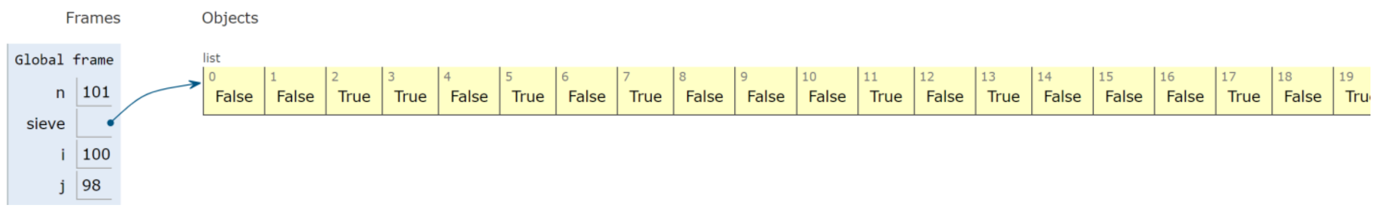
1. **Initialization**: Start with a list of Boolean values representing each number in the range, initialized to `True` (indicating potential to be a prime), except for 0 and 1, which are set to `False` because they are not prime by definition.

2. **Iterate over numbers**: Begin with the first prime number, 2, and iterate over the range. For each number (i) that is still marked as True (indicating it is a prime), proceed to mark its multiples as False (indicating they are not primes).

3. **Optimization**: The marking of non-primes starts from i^2 because, for any prime i, multiples less than i^2 (like (2i, 3i, …, (i-1)i)) would have already been marked as non-prime by smaller primes. Increment the marking by (i) (marking (i^2, i^2 + i, i^2 + 2i, …)) as these are the multiples of i.

4. **Completion**: The process continues until all numbers in the list have been considered. The numbers marked as True at the end of this process are the primes in the given range.

Below is an implementation of the Sieve of Eratosthenes in Python, designed to find all prime numbers up to a specified limit. This implementation follows the described algorithm, utilizing a list of Booleans to represent the sieve and employing optimizations to enhance efficiency.



Global frame

n	101
sieve	
i	100
j	98

list

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
False	False	True	True	False	True	False	True	False	False	False	True	False	True	False	False	False	True	False	Tru

```
n = 101  # We want to find primes up to and including 100
sieve = [True] * n  # Initialize the sieve with True values
sieve[0] = sieve[1] = False  # 0 and 1 are not primes

for i in range(2, int(n**0.5) + 1):  # Only go up to the square root of n
    if sieve[i]:  # If i is a prime
        # Set all multiples of i to False
        for j in range(i*i, n, i):  # Start from i*i, as smaller multiples would have already been marked
            sieve[j] = False

# Done! Now print the list of primes.
for i in range(n):
    if sieve[i]:
        print(i)
```

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?

Monte Carlo Method

The Monte Carlo method, named after the Monte Carlo Casino in Monaco due to its reliance on random chance, is a computational algorithm that uses repeated random sampling to obtain numerical results

The first documented application of the Monte Carlo method was in the simulations of neutron diffusion, a key process in the chain reactions required for nuclear fission. The method proved to be extremely useful in providing approximate solutions to complex mathematical problems that were otherwise unsolvable using deterministic numerical methods.

The method was formally developed by scientists working on the Manhattan Project at Los Alamos National Laboratory, with notable contributions from Stanislaw Ulam and John von Neumann. The need for a new approach arose from the complexities involved in solving problems related to the physics of nuclear reactions, which were critical for the development of the atomic bomb.The Monte Carlo method has since become a fundamental tool in various scientific fields, including physics, chemistry, finance, and engineering.

The provided code above is a simple implementation of the Monte Carlo method to approximate the value of Pi (π) and demonstrates solving problems through the use of randomness and probability. This particular implementation uses random points to estimate the area of a quarter circle, which in turn helps to approximate Pi.

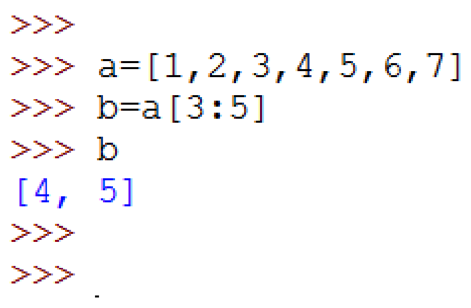
Here's a breakdown of the code and how it works:

1. **Code Overview**:
   - The **sqrt** function from the **math** module is used to calculate the square root.
   - The **random** function from the **random** module generates random floating-point numbers between 0.0 and 1.0.
   - **count** is initialized to 0 and will be used to count the number of points that fall inside the quarter circle.
   - A loop runs 1,000,000 times, each time generating a random point **(x, y)** where **x** and **y** both range from 0 to 1.
   - For each point, it checks if the point lies inside the quarter circle inscribed within the unit square by using the equation **sqrt(x*x + y*y) < 1**. If so, the **count** is incremented.
   - After all iterations, the fraction of points that fell inside the quarter circle is multiplied by 4 to approximate Pi.
2. **What it Calculates**:
   - The code is calculating an approximation of Pi (π). The value of Pi is related to the area of a circle, and by estimating the area of a quarter circle and then multiplying by 4, we can approximate the value of Pi.
3. **Theory Behind the Method**:
   - The code uses a probabilistic model to estimate the area of a quarter circle. Since the exact area of a circle with radius 1 is π, the area of a quarter circle would be π/4.
   - The unit square that bounds this quarter circle has an area of 1. By randomly generating points within this square, the proportion of points that fall inside the quarter circle should approximate the ratio of the quarter circle's area to the square's area, which is π/4.
   - Thus, by multiplying the proportion of points inside the quarter circle by 4, we get an approximation of π.
   - This method works due to the Law of Large Numbers in probability theory, which states that the results of performing the same experiment a large number of times should converge to the expected value. In this case, as the number of points increases, the approximation becomes closer to the true value of π.

Slicing lists

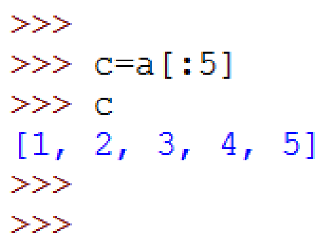
What is a slice of a list?

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.



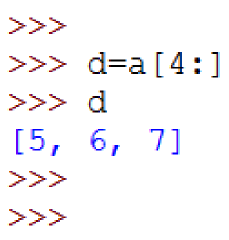
```
>>>
>>> a=[1,2,3,4,5,6,7]
>>> b=a[3:5]
>>> b
[4, 5]
>>>
>>>
```

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



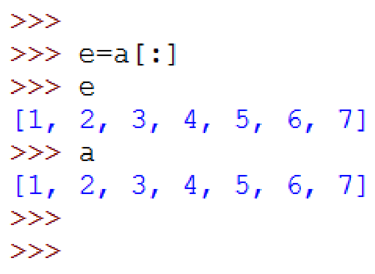
```
>>>
>>> c=a[:5]
>>> c
[1, 2, 3, 4, 5]
>>>
>>>
```

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:

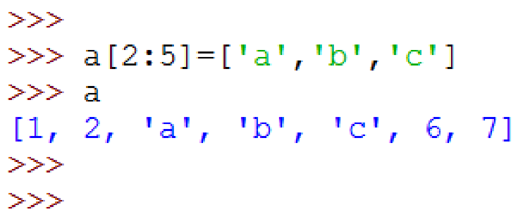


```
>>>
>>> e=a[:]
>>> e
[1, 2, 3, 4, 5, 6, 7]
>>> a
[1, 2, 3, 4, 5, 6, 7]
>>>
>>>
```

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]
>>>
>>>
```

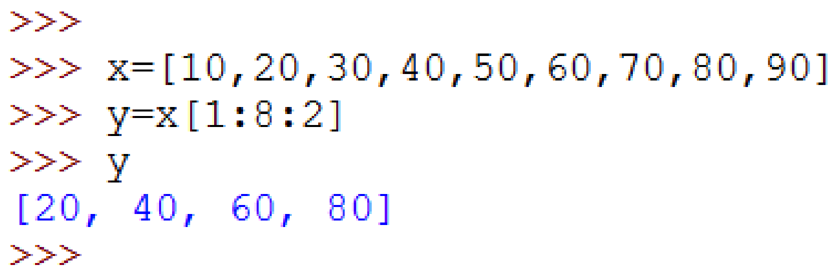
Notice that this is a **generalization** of accessing and replacing one list element as in **a[1]=12** which just replaced a single list element.

When we use a slice we can indicate a stride.

Huh?

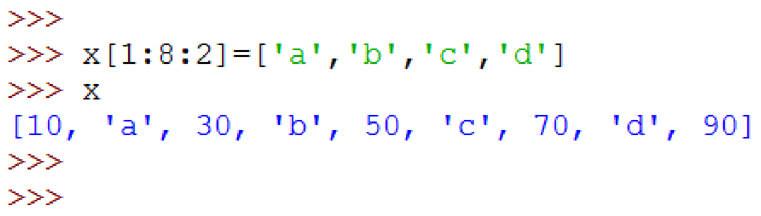
The stride is the length of the "step" that you take going from one element to the next when creating the slice.

In the following example 2 is the stride.



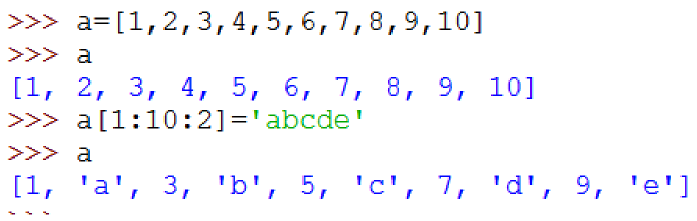
```
>>>
>>> x=[10,20,30,40,50,60,70,80,90]
>>> y=x[1:8:2]
>>> y
[20, 40, 60, 80]
>>>
```

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) and can be of any length. **But if the slice has a stride,** the list on the right hand side of the assignment mustbe the same size as the list produced by the slice..



```
>>> 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']
```

Problem:

Recall we had difficulty writing the function **delete_at_even_positions(x).**

Would something like this solve our problem?

```
d = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(d[1:len(d):2])

def delete_at_even_positions(x):
    del x[1:len(x):2]

delete_at_even_positions(d)
print(d)
```

Look at it carefully before you run the code. Then run the code to verify your analysis.

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)**)?

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).

Insertion Sort

```
def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are greater than key,
        # to one position ahead of their current position
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr


arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```

Question: Is insertion sort stable? How do you know?

Characteristics of Insertion Sort:

Best Case Scenario: The best case occurs when the array is already sorted. Here, the algorithm only makes a single comparison per element, leading to a runtime complexity of O(n).

Worst Case Scenario: The worst case occurs when the array is sorted in reverse order. Every element has to be compared with all the other elements already sorted (to its left), leading to a runtime complexity of O(n^2).

Stability: Insertion Sort is stable; it does not change the relative order of elements with equal keys.

In-Place: It is an in-place sorting algorithm, as it only requires a constant amount O(1) of additional memory space.

On line: can process its input piece-by-piece in a serial fashion, i.e., it can sort a list as it receives it, without needing the entire list beforehand..

Insertion Sort's simplicity and the fact that it is in-place and stable make it useful for small datasets and as part of more complex algorithms like **Timsort**.

Is insertion sort more efficient that selection sort? Why/how?

Insertion sort can be made more efficient by using a binary search to find the insertion point for the new element.

```
def binary_search(arr, val, start, end):
    """Find index to insert val within arr[start:end] via binary search."""
    while start < end:
        mid = (start + end) // 2
        if arr[mid] < val:
            start = mid + 1
        else:
            end = mid
    return start

def binary_insertion_sort(arr):
    for i in range(1, len(arr)):
        val = arr[i]
        # Find the position to insert the current element
        pos = binary_search(arr, val, 0, i)
        # Shift elements to the right to make space for the current element
        arr[pos+1:i+1] = arr[pos:i]
        # Insert the current element into its correct position
        arr[pos] = val
    return arr

# Example usage
arr = [37, 23, 0, 17, 12, 72, 31, 46, 100, 88, 54]
sorted_arr = binary_insertion_sort(arr)
print(sorted_arr)
```

Then, of course, there is merge sort.

Merge Sort is a classic divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves back together.

1. Divide: Split the array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge the sorted halves to produce a single sorted array.

Key Properties:

Divide-and-Conquer: Merge Sort divides the problem into subproblems, solves them independently, and combines their solutions.
Stable: It does not change the relative order of elements with equal keys.
Not In-Place: In its typical implementation, it requires additional space proportional to the array being sorted.
Time Complexity: O(n log n) in all cases (best, average, and worst).

Merge Sort is particularly well-suited for large datasets and performs consistently well due to its O(n log n) time complexity. Its main drawback is the need for additional space, making it less space-efficient compared to some in-place sorting algorithms.

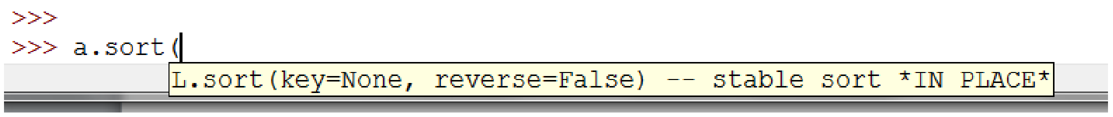
```
def merge_sort(arr):
    if len(arr) > 1:
        # Finding the mid of the array
        mid = len(arr) // 2

        # Dividing the array into two halves
        L = arr[:mid]
        R = arr[mid:]

        # Sorting the first half
        merge_sort(L)

        # Sorting the second half
        merge_sort(R)

        # Merging the sorted halves
        merge(arr, L, R)

def merge(arr, L, R):
    i = j = k = 0

    # Copy data to temp arrays L[] and R[]
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Checking if any element was left
    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1

arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print("Sorted array is:", arr)
```

- The `merge_sort` function first checks if the array has more than one element. If it does, it finds the midpoint and splits the array into two halves, `L` and `R`.
- It then recursively calls itself to sort `L` and `R` separately.
- Once `L` and `R` are sorted, it calls the `merge` function to combine these sorted halves into a single sorted array.

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

1. **sort() – is a method of the list class and is a stable 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 "<".

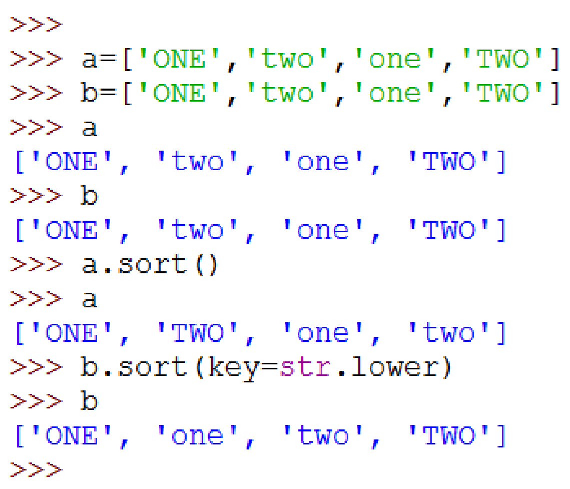
Why is it called a "keyword argument"?

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'
>>>
```

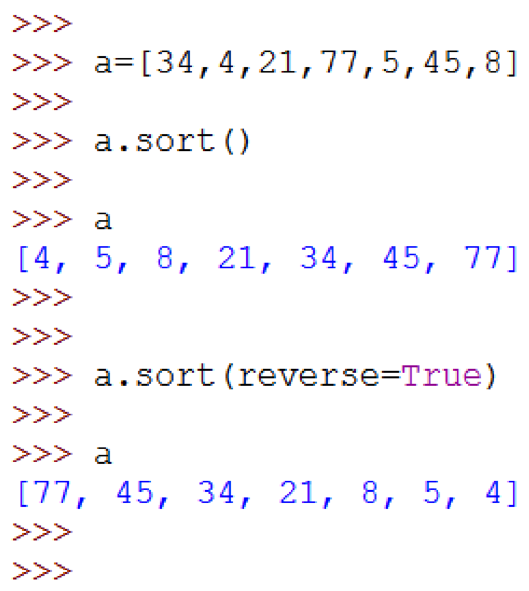
The default value for key is None.



```
>>>
>>> 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

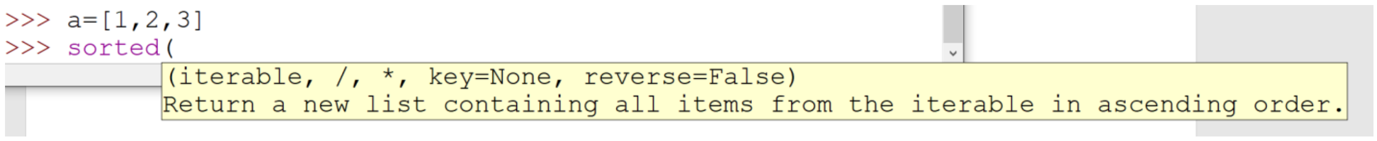
x=[1,2,3], then after it is reversed x would be [3,2,1].

Do this in two ways.

2. <u>**sorted()**</u>

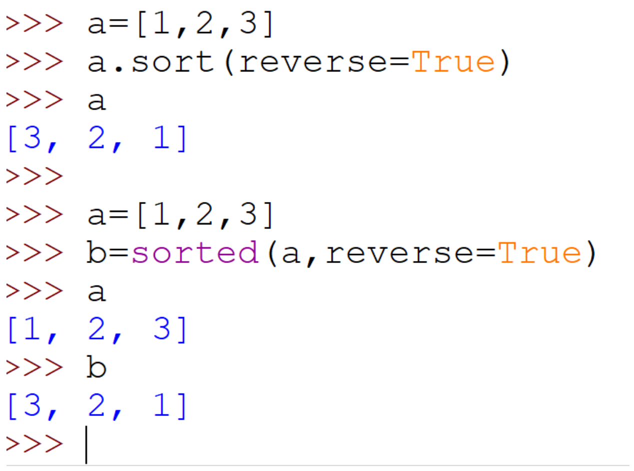
<u>**While .sort() can only be used with lists, sorted() can be used on any iterable.**</u>

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 <u>sorted()</u> function is guaranteed to be **stable**.

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 <u>we can use our own functions as well.</u>

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 "<u>lambda expression</u>."

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:

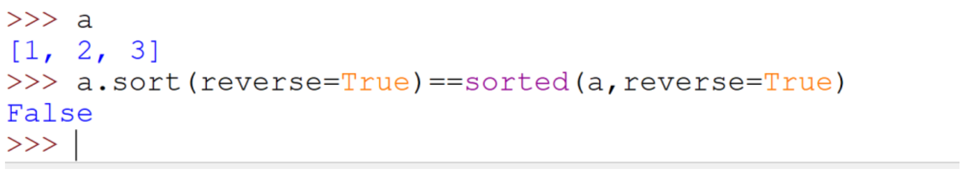
```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The key-function pattern is very common, so Python provides convenient 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
>>> 
```

Tim sort - the sort of sort Python uses in its sorts.

We studied Binary Insertion Sort and Merge Sort.

While Insertion Sort has its advantages, such as simplicity, low overhead for small arrays, and efficiency when the data is nearly sorted, Merge Sort offers significant benefits for large datasets, including consistent O(n log n) performance, stability, suitability for external sorting, and the potential for parallelization.

Timsort is a sophisticated sorting algorithm that combines the best aspects of Merge Sort and Binary Insertion Sort, designed to perform optimally on various kinds of real-world data, including those with ordered sequences (runs) and random distribution. It is the default sorting algorithm in Python (used by `sorted()` and `list.sort()`) and Java's Arrays.sort() for objects.

Core Principles:

1. **Adaptive**: Timsort adapts to the structure of the data, efficiently handling both partially ordered and random datasets by leveraging existing order.

2. **Stable**: It maintains the relative order of equal elements, crucial for multi-level sorting based on multiple attributes.

3. **Efficient for Small Data**: It uses Binary Insertion Sort for small arrays or to extend short runs, benefiting from its low overhead.

4. **Minimizes Comparisons**: Timsort employs a galloping mode in its merge function to skip over large portions of data when possible, reducing the number of comparisons.

How Timsort Works:

1. **Identifying Runs**: Timsort starts by scanning the dataset to identify naturally occurring sorted sequences (runs). These runs can be either ascending or descending; descending runs are reversed to maintain consistency.

2. **Minrun Selection**: The algorithm chooses a minimum run size (`minrun`) based on the size of the array to ensure a balance between the number of runs and their sizes. This helps in optimizing the performance across different datasets.

3. **Extending Runs**: If a run is shorter than `minrun`, Timsort uses Binary Insertion Sort to extend it. This step benefits from the insertion sort's efficiency on small datasets while ensuring that all runs meet the minimum size requirement.

4. **Merging Strategy**: Timsort maintains a stack of pending runs to be merged. It uses specific invariants (related to the sizes of the runs) to decide which runs to merge next. This adaptive approach ensures efficient merging by balancing the sizes of runs being merged and reducing the total number of merge operations.

5. **Galloping Mode**: During the merge process, if one run's elements are consistently "winning" over the other's, Timsort switches to galloping mode. This mode uses binary search to find the point up to which elements from the winning run can be merged in one go, significantly reducing the number of comparisons.

Practical Considerations:

- Timsort is designed to perform well on a wide variety of data distributions, making it a versatile and reliable sorting algorithm for real-world applications.
- Its stability and adaptiveness to the existing order in the data make it particularly useful for datasets that are not completely random, which is often the case in practical scenarios.
- The algorithm's complexity lies in managing the run stack and deciding the optimal merging strategy, which involves keeping track of the size of runs and applying the merge invariants.

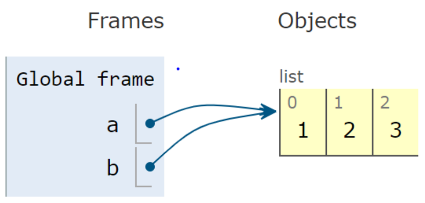
Takeaway:

Timsort's ingenious combination of Binary Insertion Sort and Merge Sort, along with its adaptive merging strategy and galloping mode, make it an outstanding general-purpose sorting algorithm. It excels in handling real-world data, offering a blend of efficiency, stability, and adaptability that is hard to match.

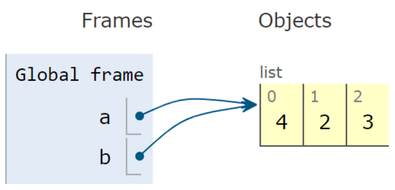
Copying lists

Let a=[1,2,3]. We want b to be a copy of a. How do we do this?

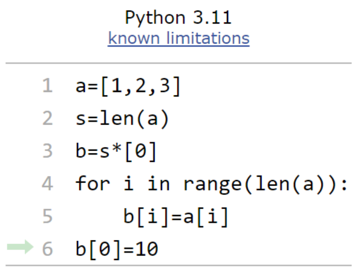
1. How about b=a?



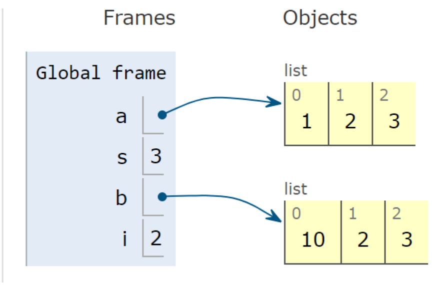
But if we the write b[0]=4, we get:



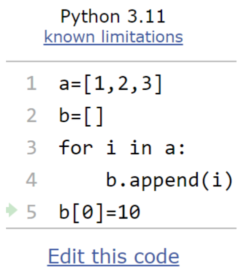
2. How about:



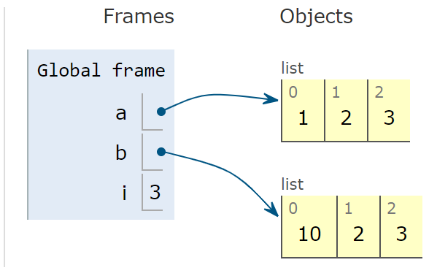
```
a=[1,2,3]
s=len(a)
b=s*[0]
for i in range(len(a)):
    b[i]=a[i]
b[0]=10
```



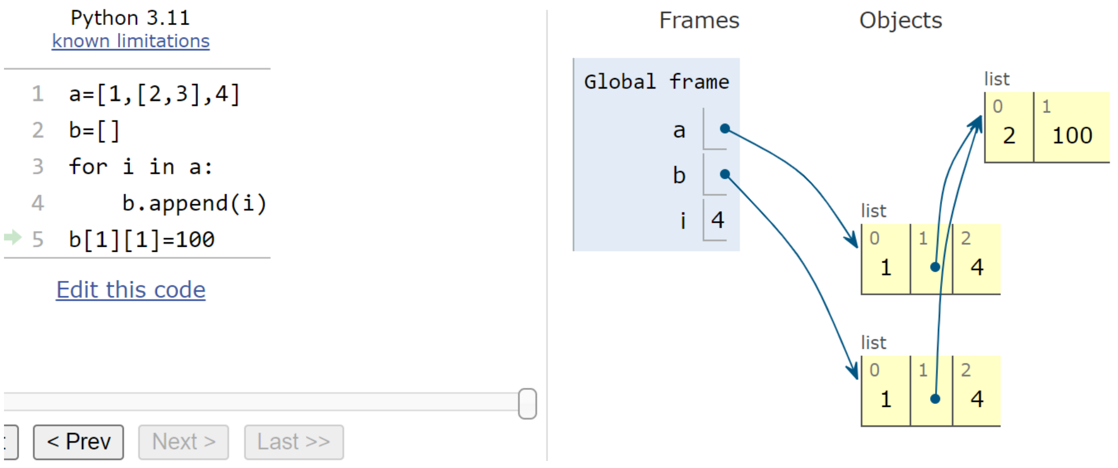
Or:



```
a=[1,2,3]
b=[]
for i in a:
    b.append(i)
b[0]=10
```



3. But what about:



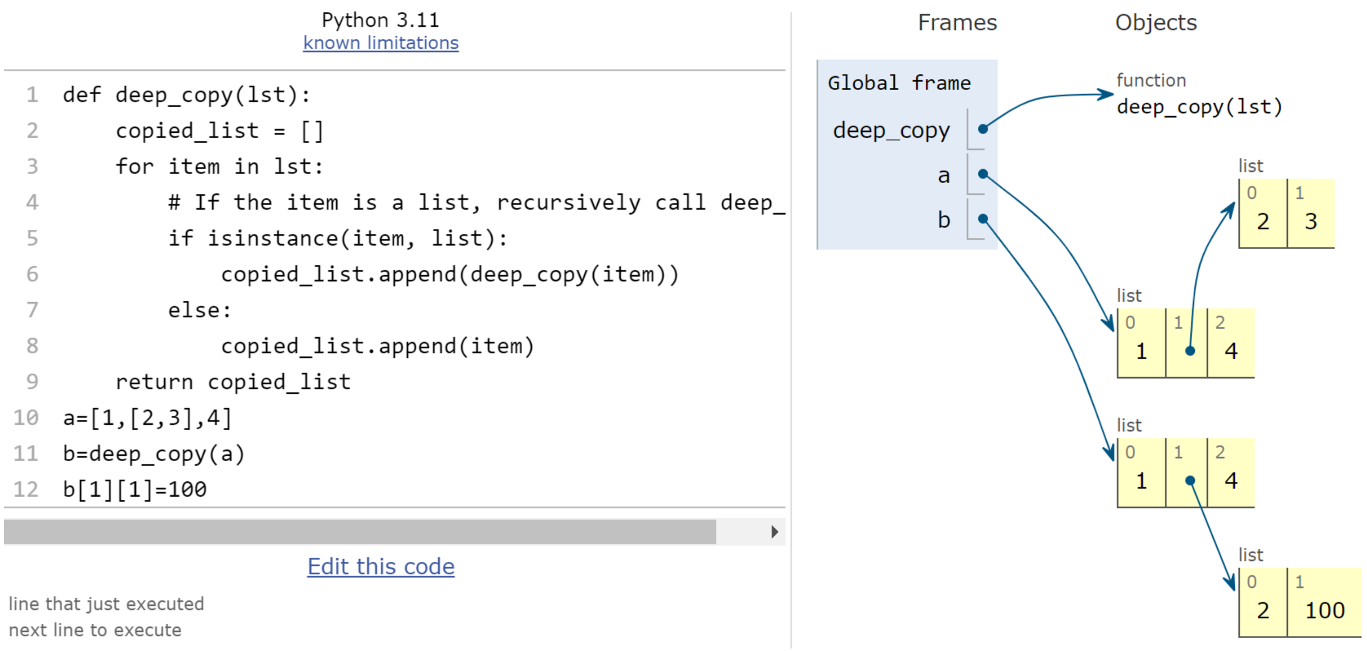
Python 3.11
known limitations

```
a=[1,[2,3],4]
b=[]
for i in a:
    b.append(i)
b[1][1]=100
```

Edit this code

Not exactly what we wanted.

4. How about:



Python 3.11
known limitations

```
def deep_copy(lst):
    copied_list = []
    for item in lst:
        # If the item is a list, recursively call deep_
        if isinstance(item, list):
            copied_list.append(deep_copy(item))
        else:
            copied_list.append(item)
    return copied_list
a=[1,[2,3],4]
b=deep_copy(a)
b[1][1]=100
```

Edit this code

line that just executed
next line to execute

This works!

But ….what about if the list elements were dictionaries, or sets, or other structures containing circular references or …. The world is complicated.

The copy module.

Enter the **copy module**. It contains **copy** (a shallow copy) and **deepcopy** (a deep copy). deepcopy is much more sophisticated than the recursive copy above. It handles all the data types and edge cases.

The difference between them lies in how they treat complex objects that contain other objects, like lists or dictionaries.

<u>Shallow Copy (`copy`)</u>

A shallow copy creates a new object, but instead of copying the nested objects, it just copies the references to them. This means the new object is a separate instance, and changes to the top-level object won't affect the copy. However, if the original object contains other objects (like a list containing other lists), changes to these nested objects will be reflected in both the original and the shallow copy because they both refer to the same nested objects. It’s like the copy we wrote above.

For example:

```
import copy

original_list = [[1, 2, 3], [4, 5, 6]]
shallow_copied_list = copy.copy(original_list)

shallow_copied_list[0][0] = 'X'  # Modifying a nested object

print(original_list)  # Output: [['X', 2, 3], [4, 5, 6]]
print(shallow_copied_list)  # Output: [['X', 2, 3], [4, 5, 6]]
```

As you can see, changing a nested element in the shallow copy also affects the original list.

Deep Copy (`deepcopy`)

A deep copy creates a new object and recursively copies all the objects it contains. Unlike a shallow copy, the deep copy doesn't just copy references to nested objects; it creates copies of the nested objects as well. Therefore, changes to any level of the copied object won't affect the original object.

For example:

```
import copy

original_list = [[1, 2, 3], [4, 5, 6]]
deep_copied_list = copy.deepcopy(original_list)

deep_copied_list[0][0] = 'X'  # Modifying a nested object

print(original_list)  # Output: [[1, 2, 3], [4, 5, 6]]
```

Here, changing a nested element in the deep copy does not affect the original list.

The takeaway:

Shallow Copy: Creates a new object but does not create copies of nested objects; it only copies their references. Changes to nested objects in the copy will affect the original.

Deep Copy: Creates a new object and recursively copies all objects contained within it, including nested objects. Changes to any part of the deep copy won't affect the original object.

The choice between shallow and deep copying depends on the complexity of the object being copied and whether changes to the copy should affect the original object.

We can write our own simple versions of copy and deepcopy:

If you want to write a custom function to shallow copy a list in Python, you can do so by creating a new list and adding all elements from the original list to it. This can be done in several ways, such as using a loop, list comprehension, or the list() constructor. Here's an example using list comprehension, which

```
def shallow_copy(lst):
   # Create a new list with the same elements as 'lst'
   copied_list = []
   for item in lst:
         copied_list.append(item)
   return copied_list
```

This function creates a shallow copy of the list, meaning that if the list contains nested lists or other mutable objects, those nested objects will not be copied; instead, both the original and copied lists will reference the same nested objects.

For a deep copy function that can handle nested lists you can use recursion:

```
def deep_copy(lst):
   copied_list = []
   for item in lst:
      # If the item is a list, recursively call deep_copy
      if isinstance(item, list):
         copied_list.append(deep_copy(item))
      else:
         copied_list.append(item)
   return copied_list

original_list = [1, [2, 3], 4, [5, [6, 7]]]
copied_list = deep_copy(original_list)

print("Original List:", original_list)
print("Copied List:", copied_list)
```

To check that in fact a deepcopy has been made, change an element in the copied list and compare it to the original one.

```
copied_list[1][0] = 'X'
print("Modified Copied List:", copied_list)
print("Original List After Modification:", original_list)
print("Original List After Modification:", original_list)
```

This deep_copy function checks each item in the input list: if the item is a list itself, it recursively copies that list; otherwise, it simply appends the item to the new list. This way, you get a new list that's a deep copy of the original, including all nested lists.

This function is specifically designed to handle nested lists, and it will not correctly handle other mutable types like dictionaries or sets. The function checks if an item is a list (isinstance(item, list)) and only then does it perform a recursive deep copy. For other types, it simply appends the item to the new list, which means dictionaries, sets, and other mutable objects would be shallow copied, sharing references between the original and copied structures.

To make the function more general and capable of deep copying structures that may include dictionaries, sets, or other mutable types, you'd need to enhance its type checking and handling..

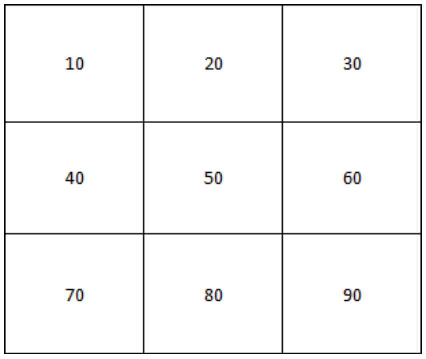
Here’s an extended version of the deep_copy function that can handle lists, dictionaries, and sets. It does not cover all edge cases or types but demonstrates how you might extend the functionality:

```
def deep_copy(obj): # this code is incomplete – a general idea of what needs to be done
    if isinstance(obj, list):
        return [deep_copy(item) for item in obj]
    elif isinstance(obj, dict):
        return {deep_copy(key): deep_copy(value) for key, value in obj.items()}
    elif isinstance(obj, set):
        return {deep_copy(item) for item in obj}
    else:
        # For simplicity, other types are not deeply copied.
        # This includes custom objects, which might require special handling.
        return obj

# Example usage with a complex structure
original_structure = [1, [2, {'a': 3, 'b': {4, 5}}, 6], 7]
copied_structure = deep_copy(original_structure)

print("Original Structure:", original_structure)
print("Copied Structure:", copied_structure)
```

Keep in mind that for a fully robust deep copy implementation, you would need to handle many more cases, and it's generally recommended to use Python's built-in copy.deepcopy() for this purpose, especially in production code or complex scenarios.

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

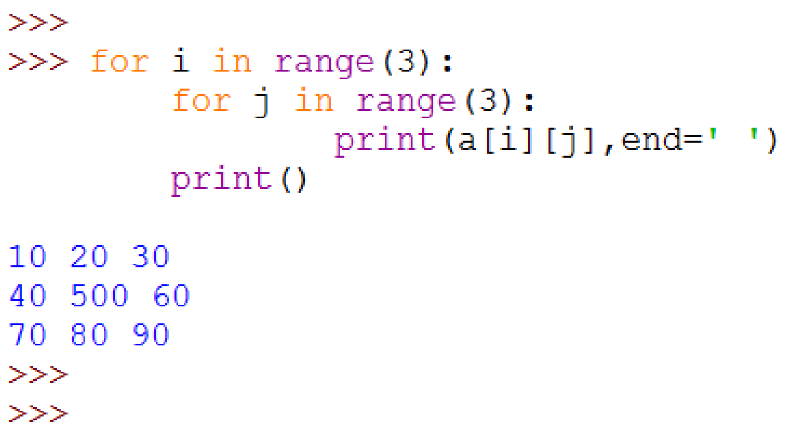
Even though a list is “really” is a list of lists, when we program its useful to think of it as a two dimensional table.

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).

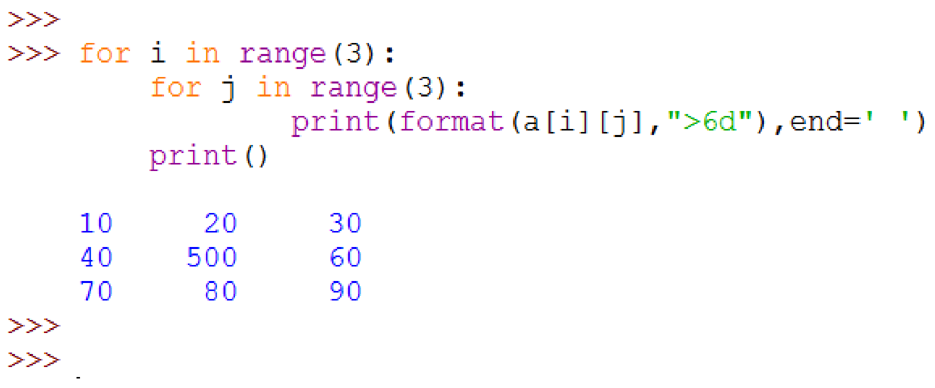
Problem:

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:

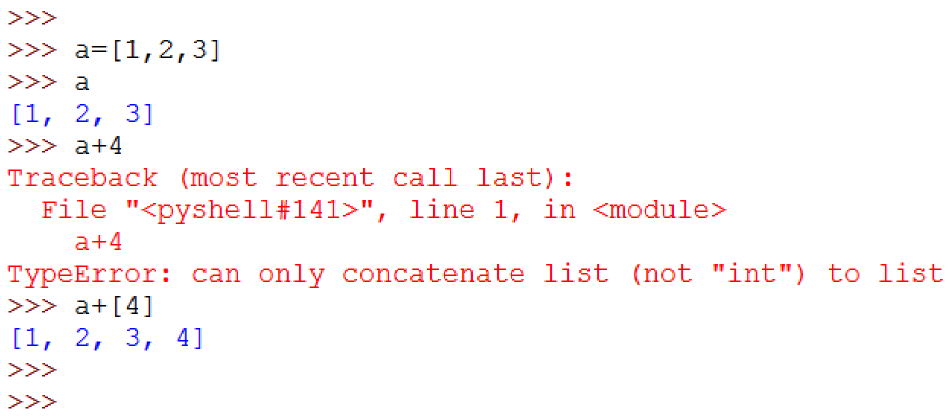
Create a 4X4 array and initialize each of the elements to 0.

Solution:

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

What does 4*[8] mean?

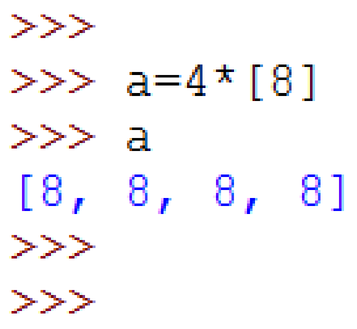
Python lets us use + and * with lists.



```
>>>
>>> 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]
>>>
>>>
```

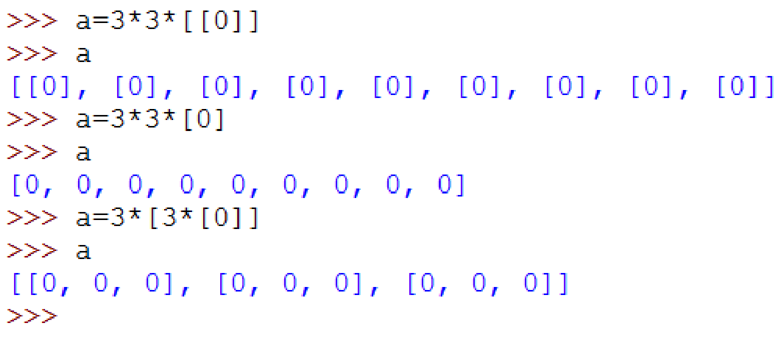
So from here you see + is **concatenate** i.e. it acts like the list function **extend**. But you can only + a list to a list, not a string to a list like you can with extend.

What about "*" ?



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

Make sure that you can explain each of the following:



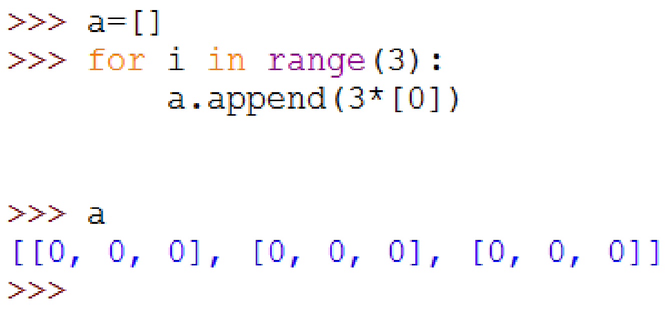
```
>>> a=3*3*[[0]]
>>> a
[[0], [0], [0], [0], [0], [0], [0], [0], [0]]
>>> a=3*3*[0]
>>> a
[0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> a=3*[3*[0]]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

The first example:

The second example:

The third example:

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



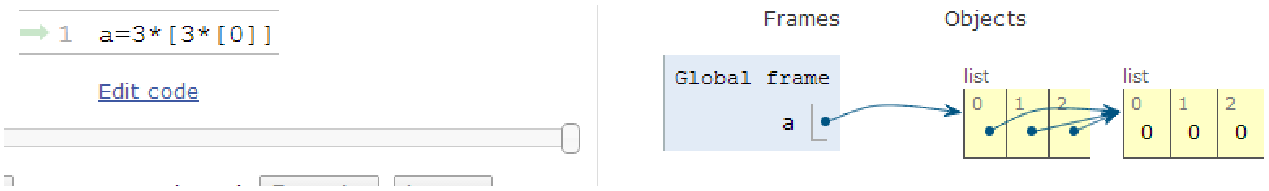
```
>>> a=[]
>>> for i in range(3):
        a.append(3*[0])


>>> 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.

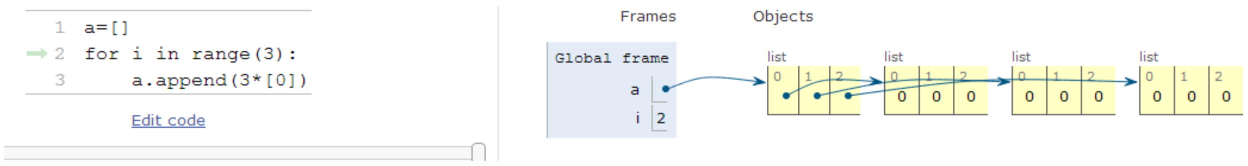
The first one produces the following when it runs:



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

What are the implications of this?

The second one, however, produces this:



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

What are the implications of this?