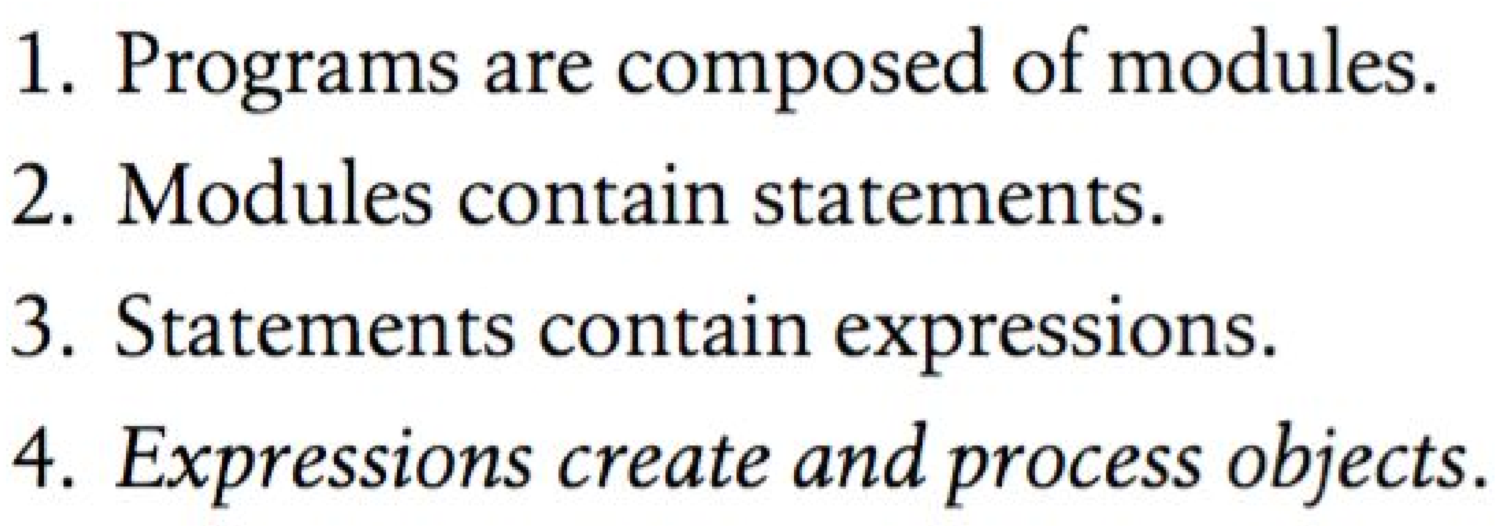
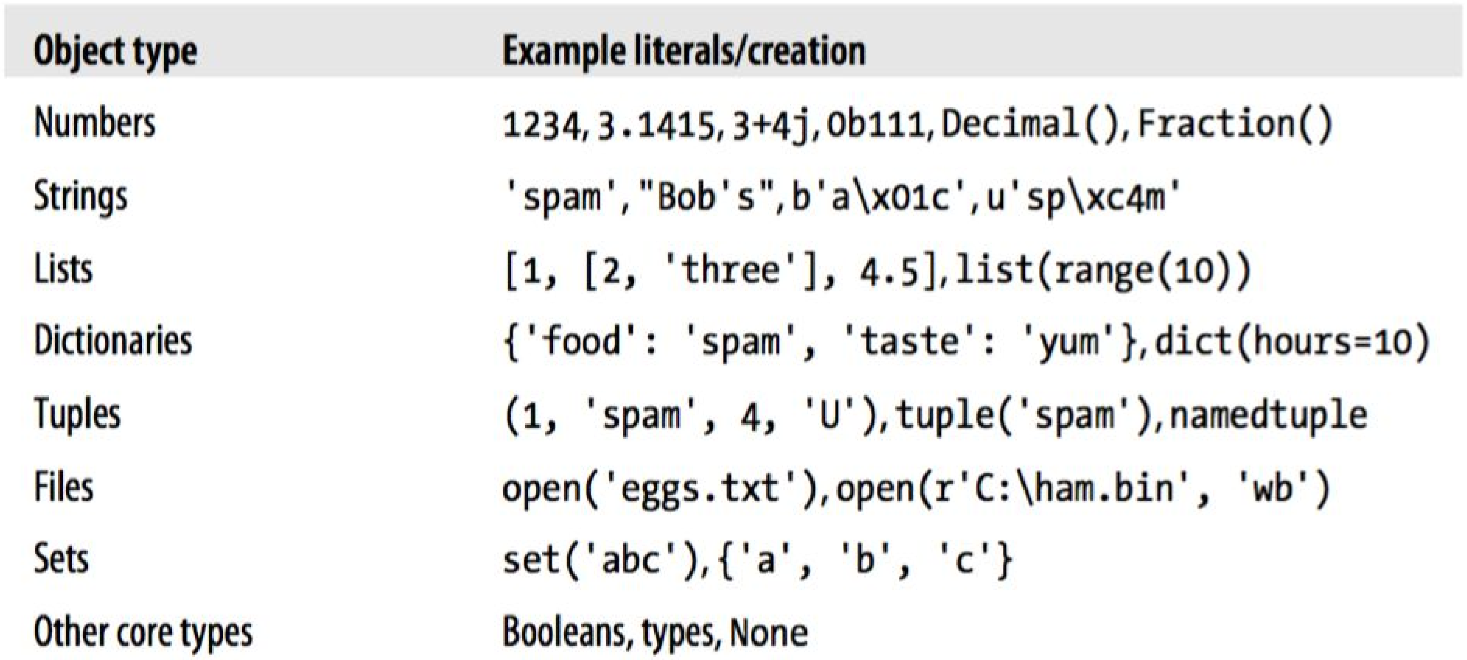
Introduction to Python

Python Conceptual Hierarchy



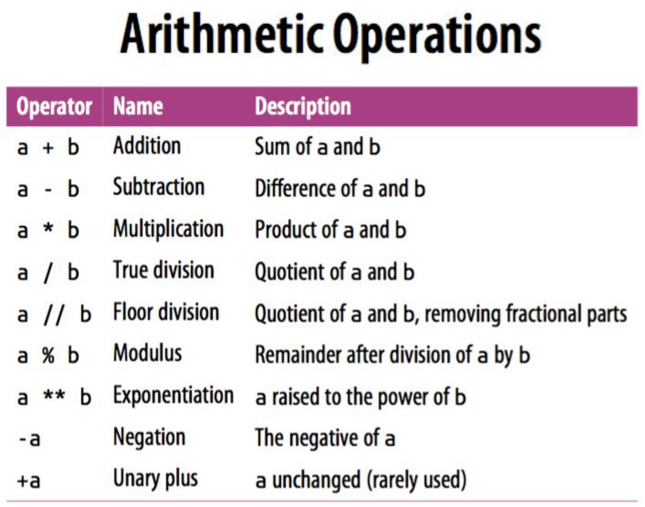
1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Data Types



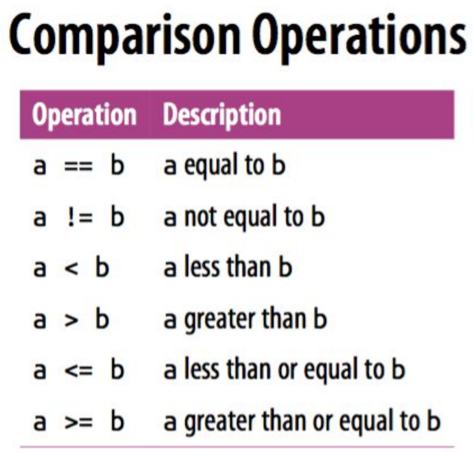
Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None

Operators



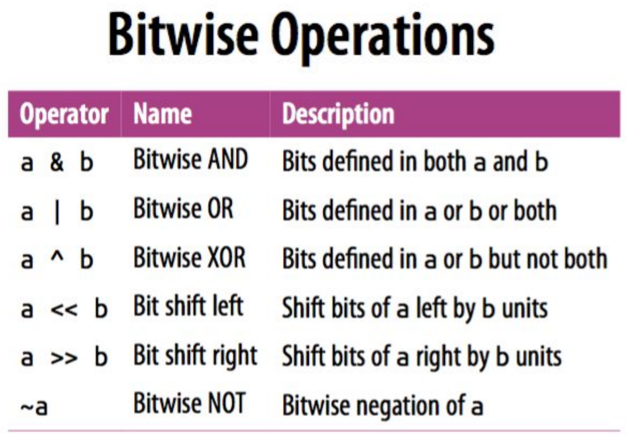
Arithmetic Operations

Operator	Name	Description
a + b	Addition	Sum of a and b
a - b	Subtraction	Difference of a and b
a * b	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
a // b	Floor division	Quotient of a and b, removing fractional parts
a % b	Modulus	Remainder after division of a by b
a ** b	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)



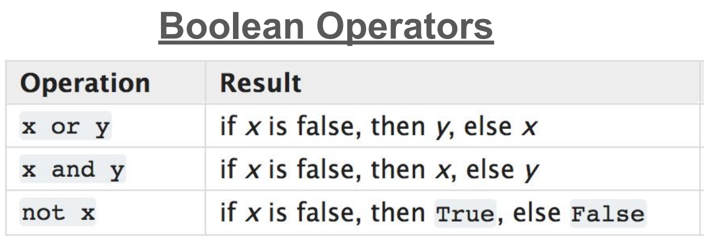
Comparison Operations

Operation	Description
a == b	a equal to b
a != b	a not equal to b
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b



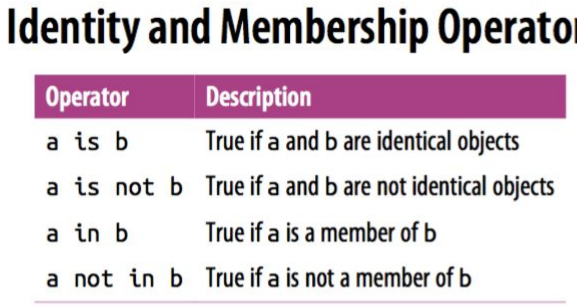
Bitwise Operations

Operator	Name	Description
a & b	Bitwise AND	Bits defined in both a and b
a | b	Bitwise OR	Bits defined in a or b or both
a ^ b	Bitwise XOR	Bits defined in a or b but not both
a << b	Bit shift left	Shift bits of a left by b units
a >> b	Bit shift right	Shift bits of a right by b units
~a	Bitwise NOT	Bitwise negation of a



Boolean Operators

Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`



Identity and Membership Operators

Operator	Description
a is b	True if a and b are identical objects
a is not b	True if a and b are not identical objects
a in b	True if a is a member of b
a not in b	True if a is not a member of b

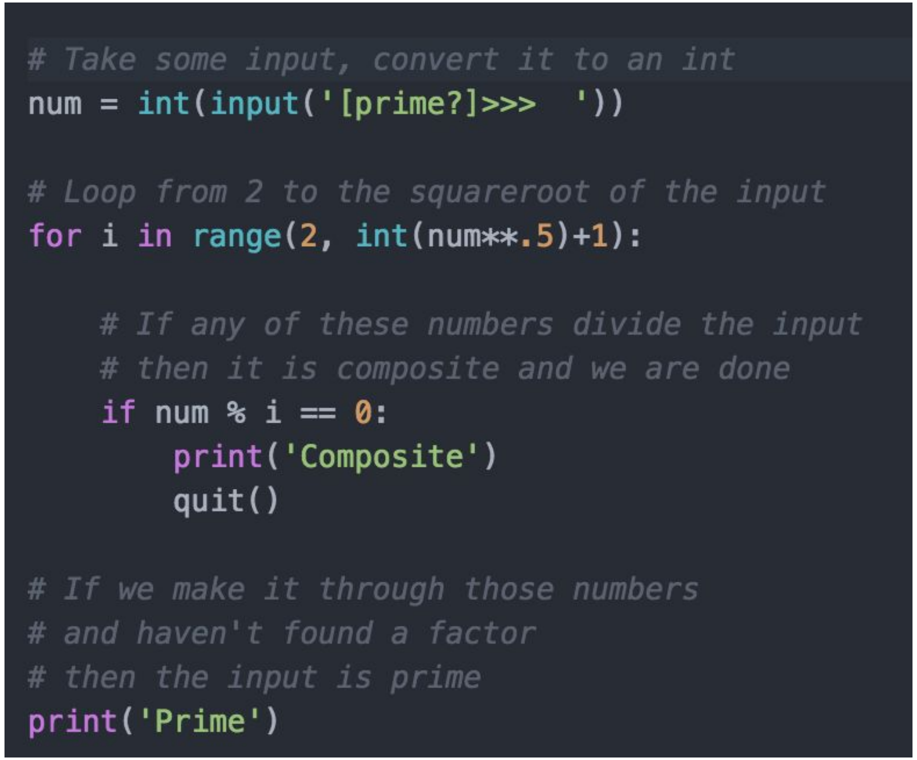
Programming - Simple Primality Testing

One of the simplest primality test is *trial division*:

> Given an input number *n*, check whether any integer *m* from 2 to √n evenly divides *n* (the division leaves no remainder).
>
> If *n* is divisible by any *m* then *n* is composite, otherwise it is prime.

Task: Write a script that asks the user for a number and then uses trial division to test if that number is prime.

Programming - Simple Primality Testing



```python
# Take some input, convert it to an int
num = int(input('[prime?]>>>  '))

# Loop from 2 to the squareroot of the input
for i in range(2, int(num**.5)+1):

    # If any of these numbers divide the input
    # then it is composite and we are done
    if num % i == 0:
        print('Composite')
        quit()

# If we make it through those numbers
# and haven't found a factor
# then the input is prime
print('Prime')
```

File I/O

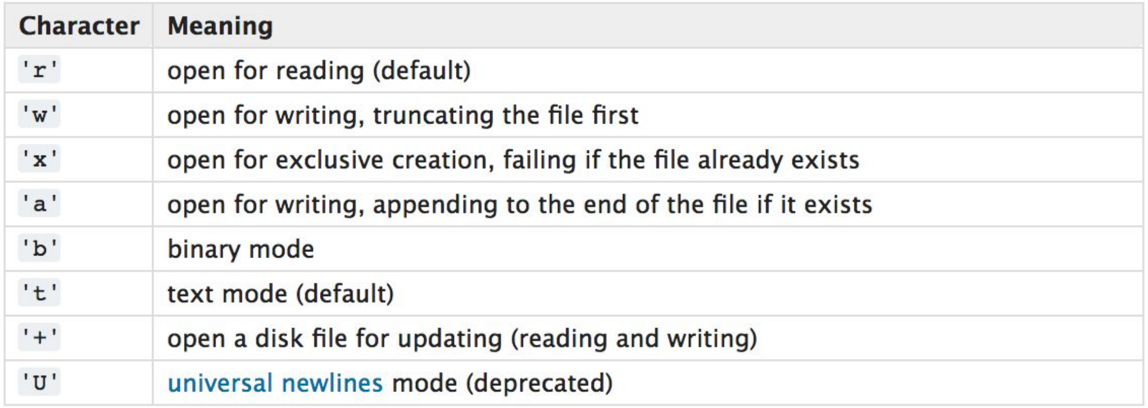
Before you can read or write a file, you have to open it using Python's built-in ***open()*** function.

`open(filename, mode)`.

This function creates a **file object**, which would be utilized to call methods associated with it.

It is most commonly used with two arguments.

- *filename* is a path-like object giving the pathname of the file to be opened.
- *mode* is an optional string that specifies the mode in which the file is opened. It defaults to `'r'`.



Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newlines mode (deprecated)

File I/O

To read a file’s contents call:

- *fileobject*.read(size)
  which reads the specified quantity of data and returns it as a string .
  - *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned.

- *fileobject*.readline()
  reads a single line from the file.

- For reading lines from a file, you can loop over the file object.
  - This is memory efficient, fast, and leads to simple code:

```
>>> for line in fileobject:
...     print(line)
...
This is the first line of the file.
Second line of the file
```

File I/O

- *fileobject*.seek(offset, reference point)
  moves to the position that is computed from adding *offset* to a *reference point*.

  - A *reference point* of 0 measures from the beginning of the file
  - 1 uses the current file position
  - 2 uses the end of the file as the reference point
  - *reference point* can be omitted and defaults to 0.

- *fileobject*.write(**string**)
  writes the contents of ***string*** to the file, returning the number of characters written.

File I/O

- When you want to save more complex data types like nested lists and dictionaries Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation).

- The standard module called json can take Python data hierarchies, and convert them to string representations. This process is called *serializing*.

- Reconstructing the data from the string representation is called *deserializing*.

File I/O

- If you have an object x, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a text file. So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

- To decode the object again, if f is a text file object which has been opened for reading:

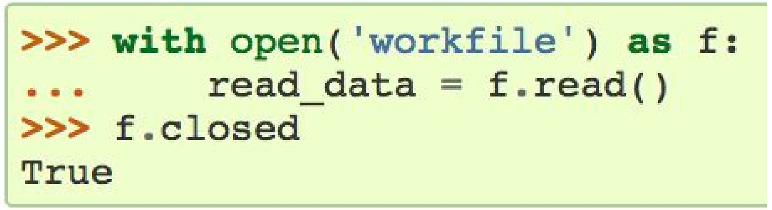
```
x = json.load(f)
```

- This simple serialization technique can handle lists and dictionaries and allows us to store these structures for later use.

File I/O

- f.close()
  closes the file and frees up any system resources used by it.
  - If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while.

- It is good practice to use the **with** keyword when dealing with file objects.
  - The advantage is that the file is properly closed after its block finishes.



```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

Functions

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**:
  - programs are lists of instructions that tell the computer what to do with the program’s input.
  - C, Pascal, and even Unix shells are procedural languages.

- In **declarative** languages:
  - You write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently.
  - SQL is the declarative language.

- **Functional** programming:
  - Decomposes a problem into a set of functions.
  - In a functional program, input flows through a set of functions.
  - Each function operates on its input and produces some output.

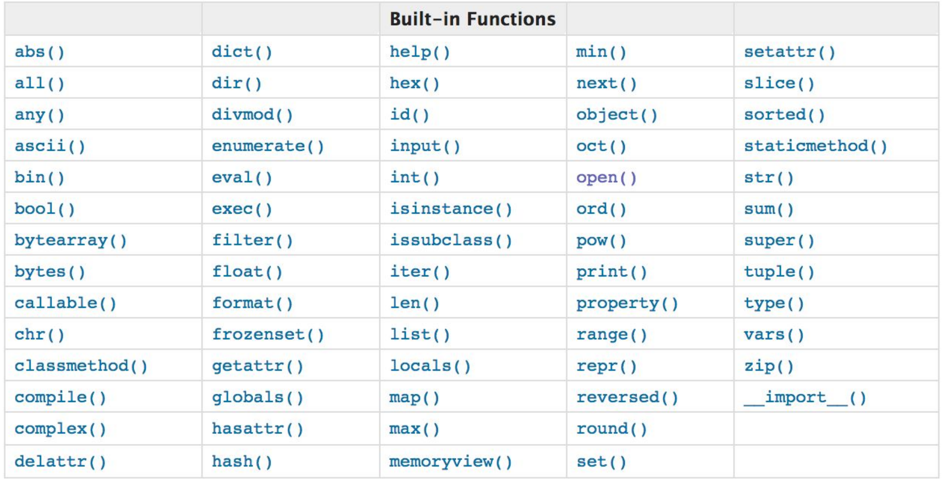
Functions

Python gives you many built-in functions but you can also create your own functions. These functions are called *user-defined functions.*

A function is a block of organized, reusable code. They group a set of statements so they can be run more than once in a program

Functions provide better modularity for your application and a high degree of code reusing.

Functions also provide a tool for splitting systems into pieces that have well-defined roles.



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Functions

def Statements

- The def statement creates a function object and assigns it a name.
  Its general form:

```
def name(arg1, arg2,... argN):
    statements
```

- Function header begins with the keyword **def** followed by the **function name** and **parentheses ( )**.
- Any **input parameters** or **arguments** should be placed within these parentheses.
- The code block within every function starts with a **colon :** and is **indented**.

Functions

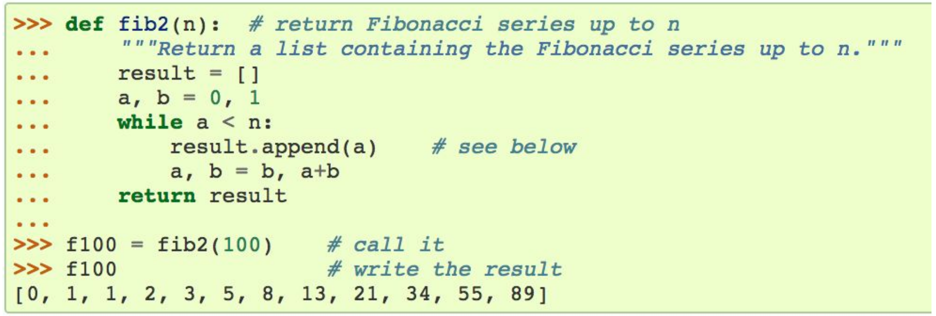
def Statements

- **def** is an executable statement - it creates a new function object at runtime (as opposed to a compiled language)
- Like all other statements **def** statements can appear where statements are valid.
  - Nested function declarations are allowed

```
if test:
    def func():                # Define func this way
        ...
else:
    def func():                # Or else this way
        ...
...
func()                         # Call the version selected and built
```

```
othername = func               # Assign function object
othername()                    # Call func again
```

Functions



```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

The keyword **def** introduces a **function definition**.

When Python reaches and runs this **def**, it creates a new function object that packages the function's code and assigns the object to the name **fib2** .

The function definition **DOES NOT** execute the function body.

To execute a function it must be **called**.

- You cannot call a function until after you have defined it.

You can **call** (run) the function in your program by adding parentheses after the function's name.

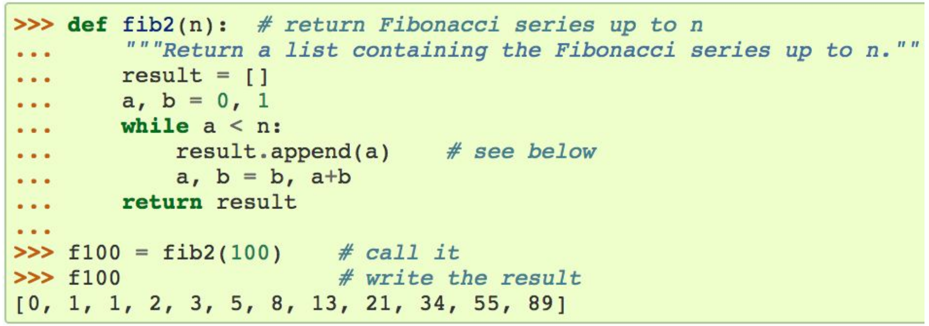
- The parentheses may optionally contain one or more object arguments

Functions

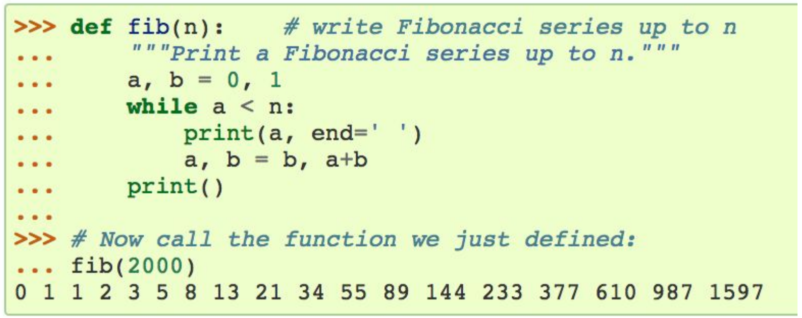
return leaves the current function call with the expression list.

Even functions without a **return** statement do return a value, None.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or docstring; it’s good practice to include docstrings in code that you write, so make a habit of it.



```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```



```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Functions

argument

A value passed to a function when calling the function. There are two kinds of argument:

- ***keyword argument***: an argument preceded by an **identifier** (e.g. `name=`) in a function call or be passed as values in a **dictionary preceded by** `**`.
    - For example, `3` and `5` are both keyword arguments in the following calls to complex():
      `complex(real=3, imag=5)`

- ***positional argument***: an argument that is not a keyword argument. Positional arguments can appear at the **beginning** of an argument list or be passed as elements of an **iterable preceded by** `*`.
    - For example, `3` and `5` are both positional arguments in the following calls:
      `complex(3, 5)`

Functions

Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
```

Keyword Arguments

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
```

Arbitrary Argument Lists

```
def write_multiple_items(file, separator, *args):
```

Functions

Default Argument Values

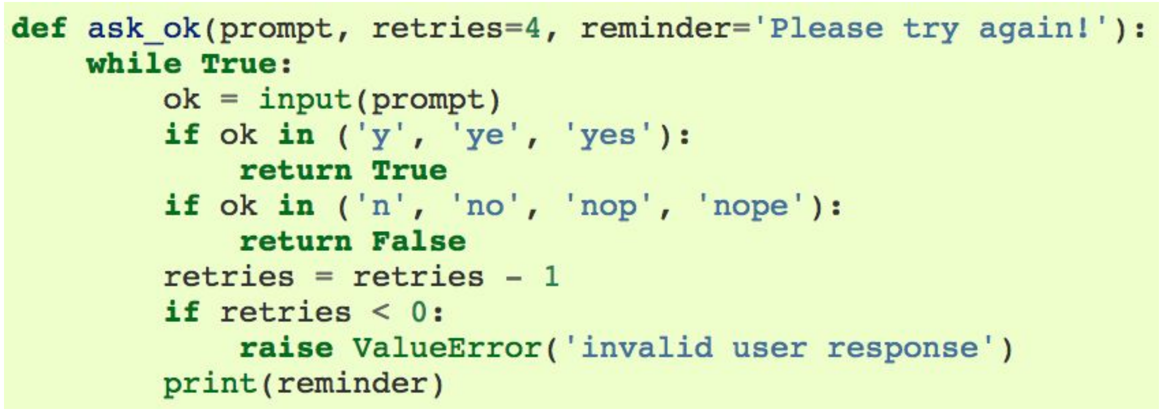
When one or more parameters have the form *parameter* `=` *expression*, the function is said to have “default parameter values.”

For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted.

If a parameter has a default value, all following parameters must also have a default value.

Functions

Default Argument Values



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

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

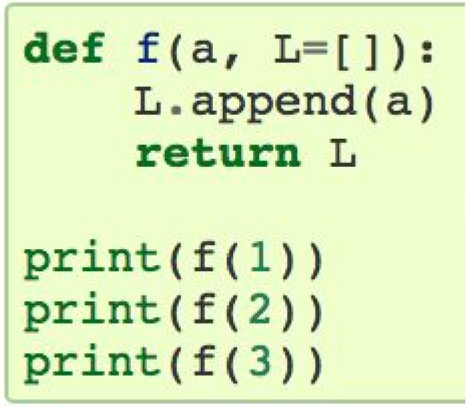
Functions

Default Argument Values

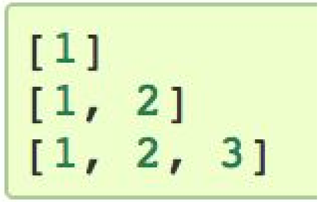
Default parameter values are evaluated from left to right when the function definition is executed.

The default value is evaluated only once.

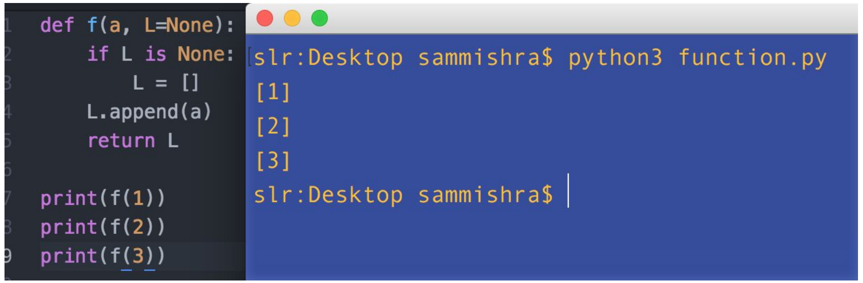
This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.



```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```



```
[1]
[1, 2]
[1, 2, 3]
```



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

print(f(1))
print(f(2))
print(f(3))
```

```
slr:Desktop sammishra$ python3 function.py
[1]
[2]
[3]
slr:Desktop sammishra$
```

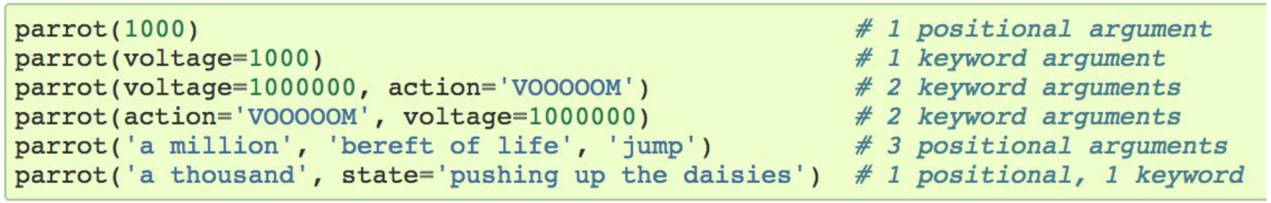
Functions

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type).:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

This function can be called in any of the following ways:



```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

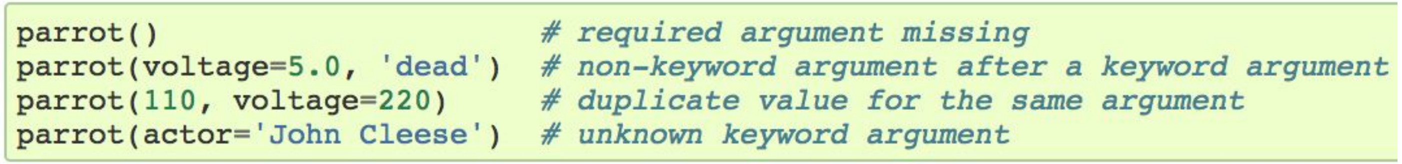
Functions

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type).:

```python
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

All the following calls would be invalid:



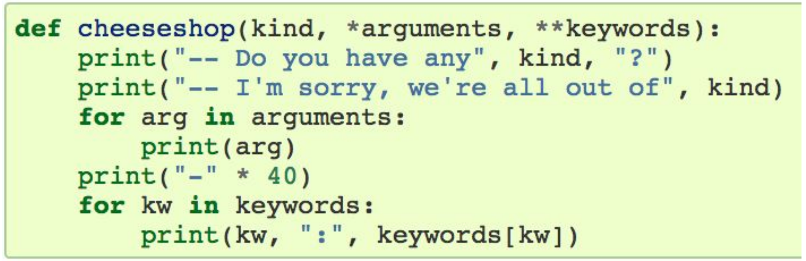
```python
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Functions

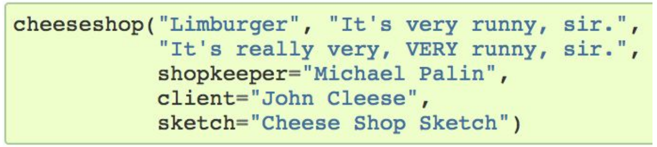
Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments.

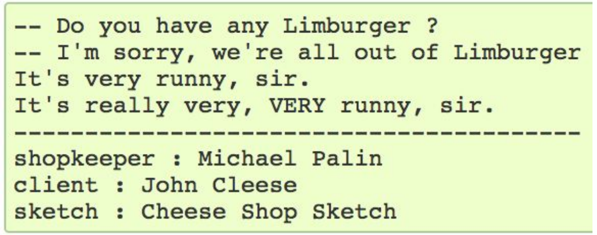
- Formal parameter of the form `**name` - receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.
- Formal parameter of the form `*name` - receives a tuple containing the positional arguments beyond the formal parameter list.



```python
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```



```python
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```



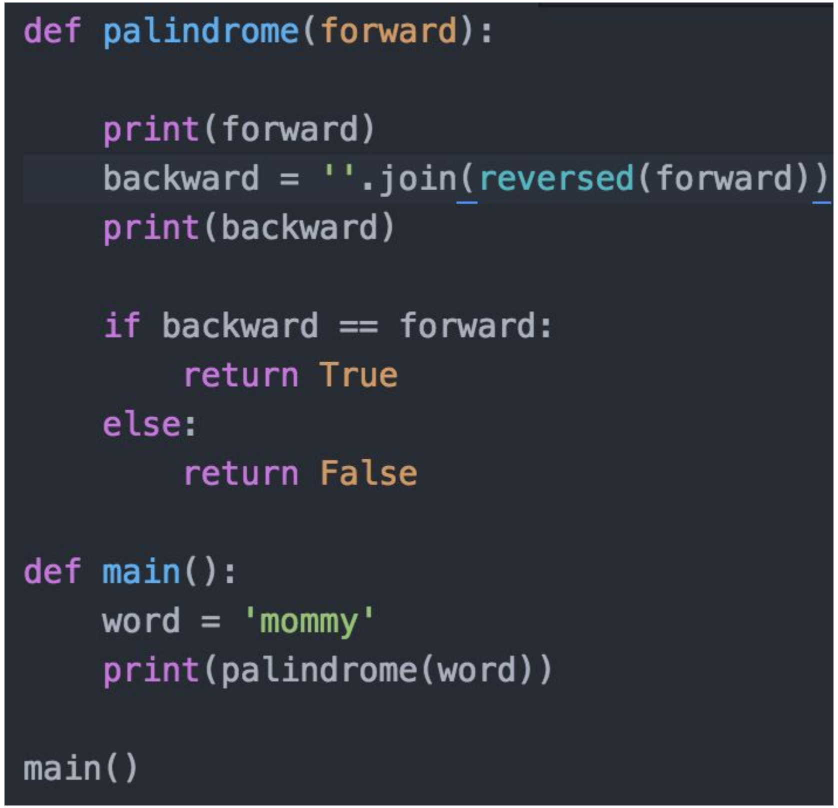
```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Palindromes

A **palindrome** is a word which reads the same backward as forward, such as *madam* or *racecar*.

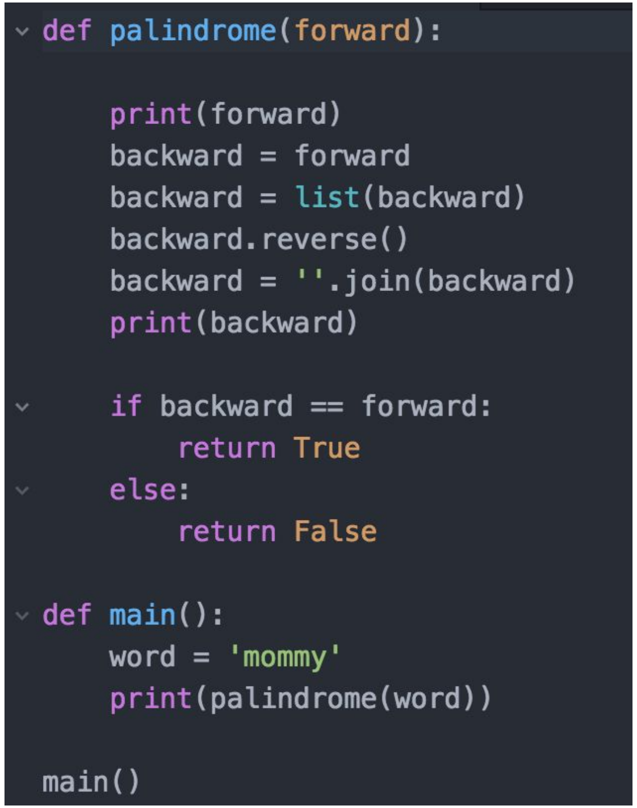
Task: Given a word as input determine if it is a palindrome. Return True if the word is a palindrome, False if it is not.

Palindromes



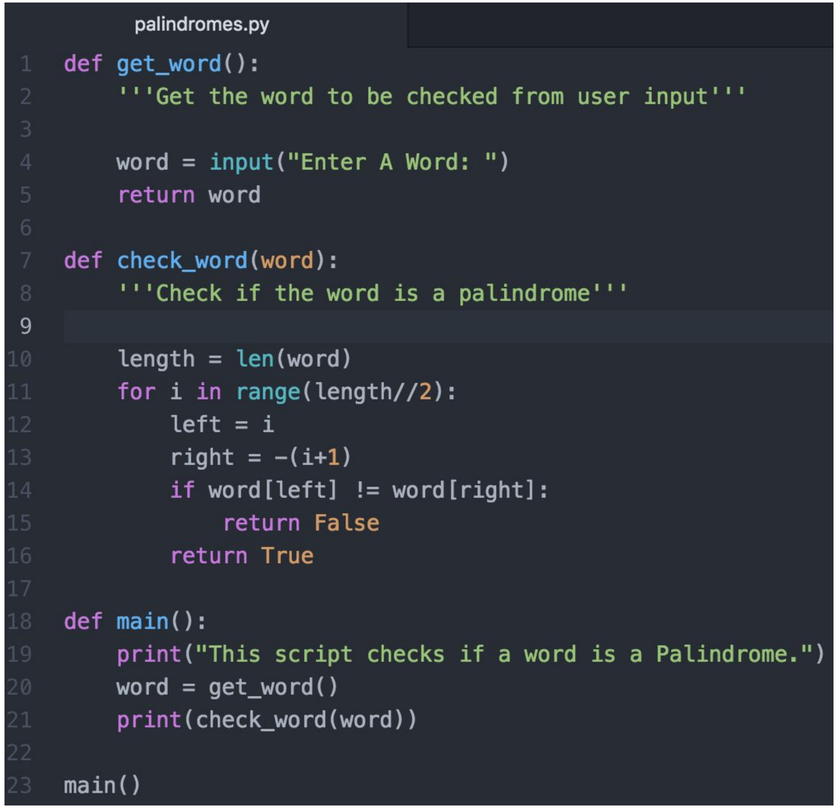
```python
def palindrome(forward):

    print(forward)
    backward = ''.join(reversed(forward))
    print(backward)

    if backward == forward:
        return True
    else:
        return False

def main():
    word = 'mommy'
    print(palindrome(word))

main()
```

Palindromes



```python
def palindrome(forward):

    print(forward)
    backward = forward
    backward = list(backward)
    backward.reverse()
    backward = ''.join(backward)
    print(backward)

    if backward == forward:
        return True
    else:
        return False

def main():
    word = 'mommy'
    print(palindrome(word))

main()
```

Palindromes


palindromes.py

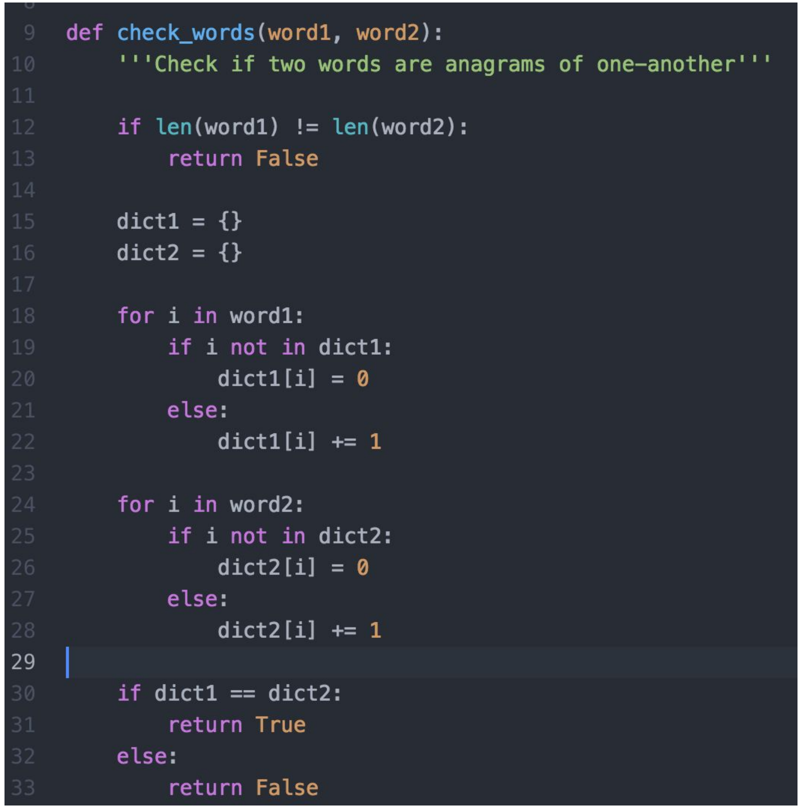
```python
def get_word():
    '''Get the word to be checked from user input'''

    word = input("Enter A Word: ")
    return word

def check_word(word):
    '''Check if the word is a palindrome'''

    length = len(word)
    for i in range(length//2):
        left = i
        right = -(i+1)
        if word[left] != word[right]:
            return False
        return True

def main():
    print("This script checks if a word is a Palindrome.")
    word = get_word()
    print(check_word(word))

main()
```

Anagrams

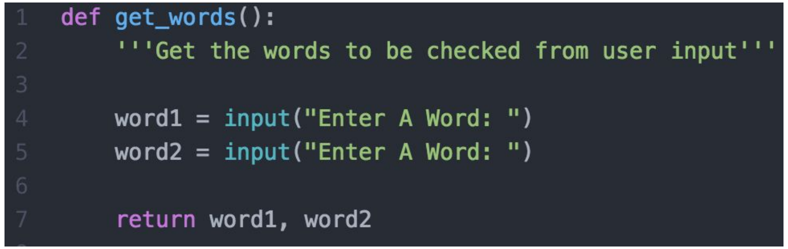
An **anagram** is the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example, the word anagram can be rearranged into "nag a ram".

Task: Given two words as input determine if they are an anagram of one-another. Return True if the word is an anagram, False if it is not.

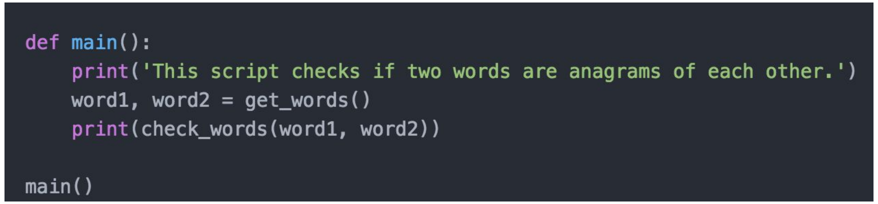
Anagrams



```python
def check_words(word1, word2):
    '''Check if two words are anagrams of one-another'''

    if len(word1) != len(word2):
        return False

    dict1 = {}
    dict2 = {}

    for i in word1:
        if i not in dict1:
            dict1[i] = 0
        else:
            dict1[i] += 1

    for i in word2:
        if i not in dict2:
            dict2[i] = 0
        else:
            dict2[i] += 1

    if dict1 == dict2:
        return True
    else:
        return False
```



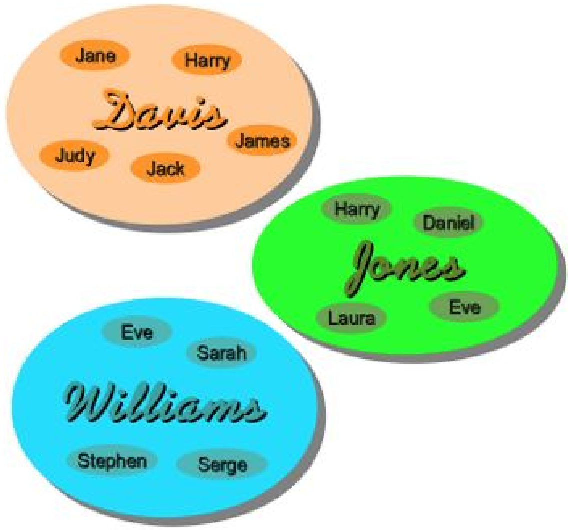
```python
def get_words():
    '''Get the words to be checked from user input'''

    word1 = input("Enter A Word: ")
    word2 = input("Enter A Word: ")

    return word1, word2
```



```python
def main():
    print('This script checks if two words are anagrams of each other.')
    word1, word2 = get_words()
    print(check_words(word1, word2))

main()
```

Variable Scope

- *A* **namespace** is a mapping from names to objects.
- Generally speaking, a namespace is a naming system for making names unique to avoid ambiguity.
    - A namespacing system from daily life - the naming of people in firstname and surname.



Variable Scope

- Many programming languages use **namespaces** for **identifiers**.
    - In a single Python program there are multiple namespaces.
    - An identifier defined in a particular namespace is **bound**/associated with that namespace.
    - This way, the same identifier can be independently defined in multiple namespaces.

- Examples of namespaces:
    - the set of built-in names
    - the global names in a module
    - the local names in a function invocation
    - In a sense the set of attributes of an object also form a namespace

- **There is absolutely no relation between names in different namespaces**

Variable Scope

Lifetime of a Namespace

Not every namespace is accessible (or alive) at any moment during the execution of the script.

- The **built-in namespace** is present from beginning to end.
- The **global namespace** of a module is generated when the module is read in.
- The **module namespaces** normally last from when it is imported until the script ends.
- The **local namespace of a particular function** is created when the function is called. The namespace is destroyed either if the function ends, i.e. returns, or if the function raises an exception.

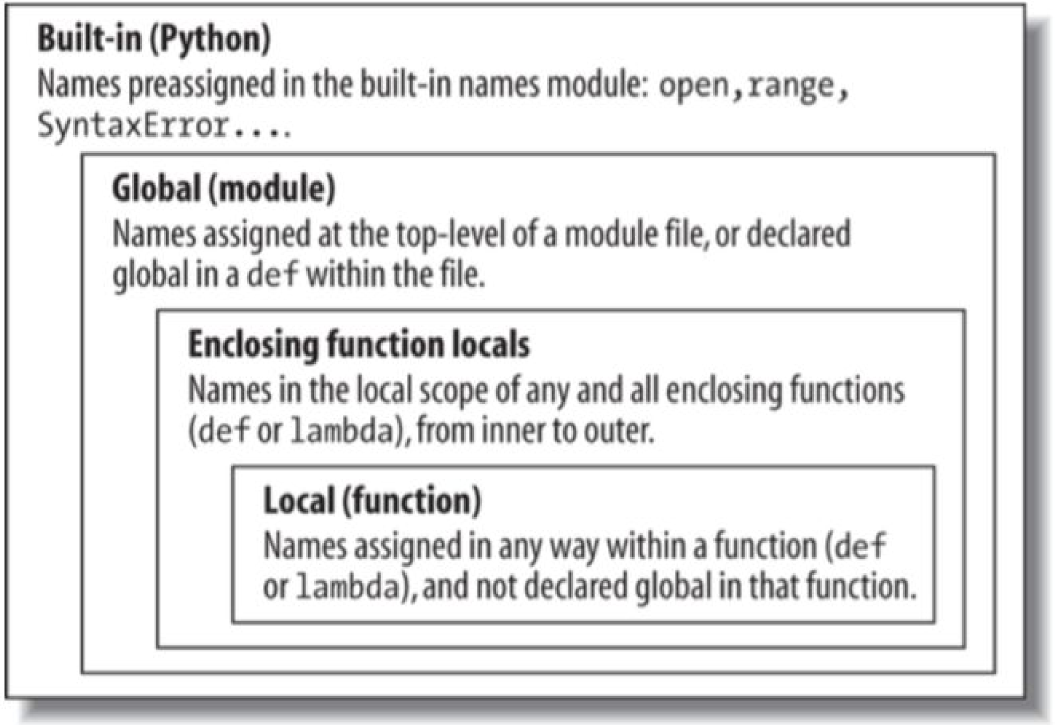
Variable Scope

Scopes
The scope of a namespace is the area of a program where this name can be unambiguously used.

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible. Python works its way out to resolve which scope you are referring to.

- the innermost scope i.e. the current and alive function, which is searched first, contains the **local names**
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains **non-local, but also non-global names**
- the next-to-last scope is searched next and it contains the current module’s **global names**
- the outermost scope, which is searched last, is the namespace containing **built-in names.**

Variable Scope



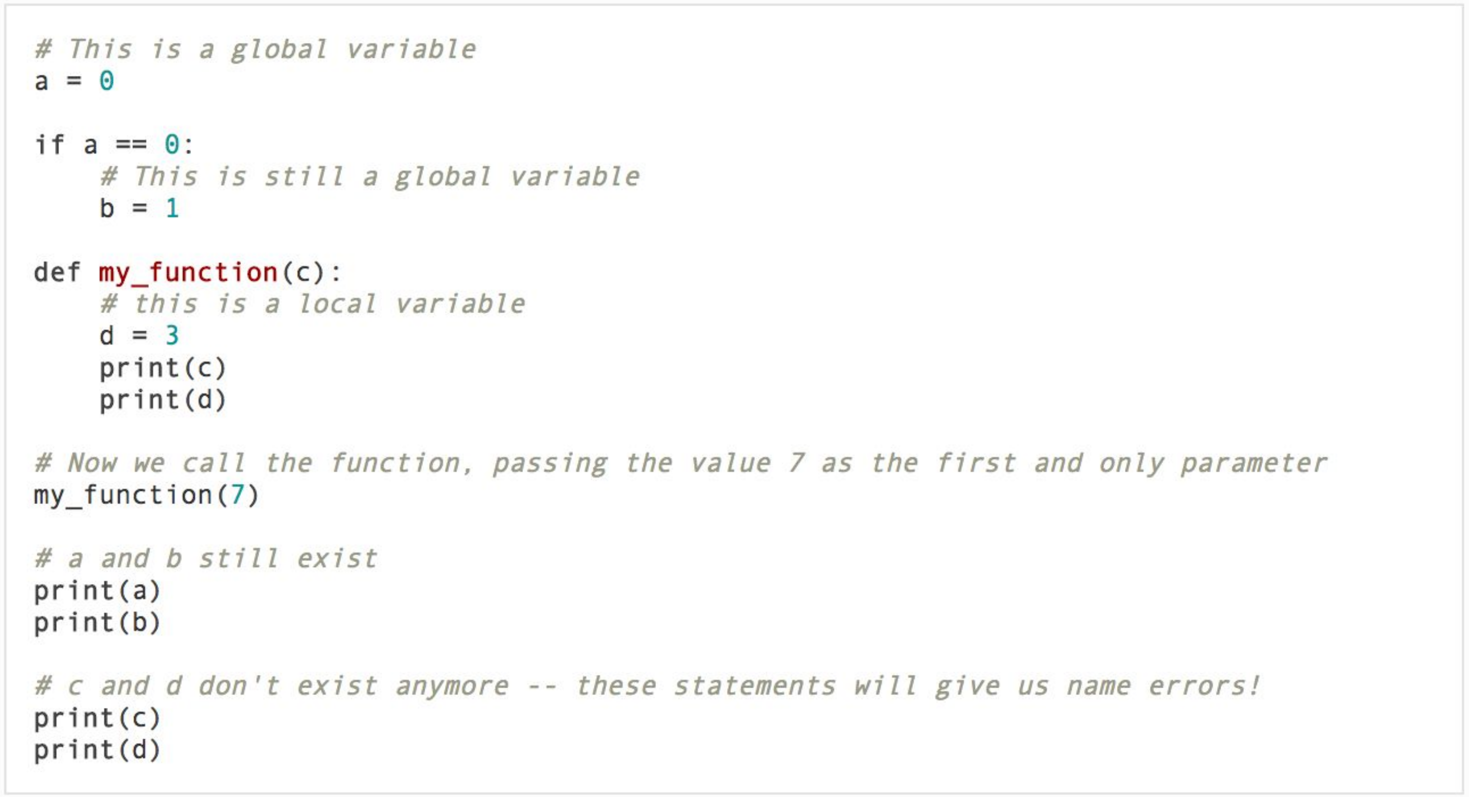
Built-in (Python)
Names preassigned in the built-in names module: `open`, `range`, `SyntaxError`....

Global (module)
Names assigned at the top-level of a module file, or declared global in a `def` within the file.

Enclosing function locals
Names in the local scope of any and all enclosing functions (`def` or `lambda`), from inner to outer.

Local (function)
Names assigned in any way within a function (`def` or `lambda`), and not declared global in that function.

Variable Scope



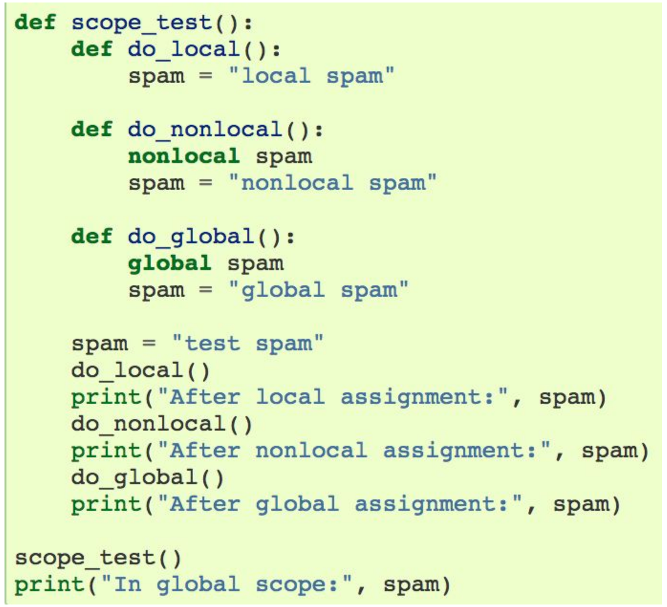
```python
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

Variable Scope

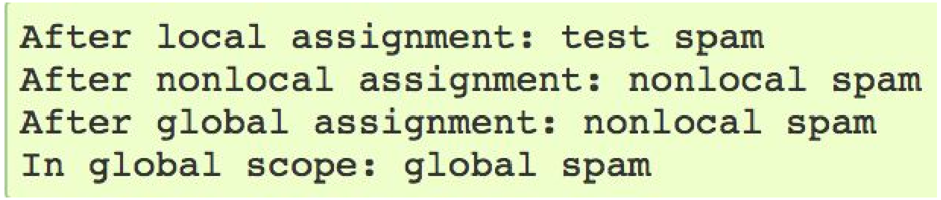
- If a name is declared **global**, then all references and assignments go directly to the middle scope containing the module’s global names.

- To rebind variables found outside of the innermost scope, the **nonlocal** statement can be used
  - if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Variable Scope



```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```



```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn't change *scope_test*'s binding of *spam*.

The nonlocal assignment changed *scope_test*'s binding of *spam*

The global assignment changed the module-level binding.

You can also see that there was no previous binding for *spam* before the global assignment.

Functions

Recursion

Sometimes it is difficult to define an object explicitly.

However,
It may be easy to define this object in terms of itself.

Functions

Recursion

Termination Condition:

A recursive function has to **terminate** to be correct and usable.
A recursive function terminates, if with every recursive call the solution of the problem is downsized (becomes easier) and moves towards a **base case**.

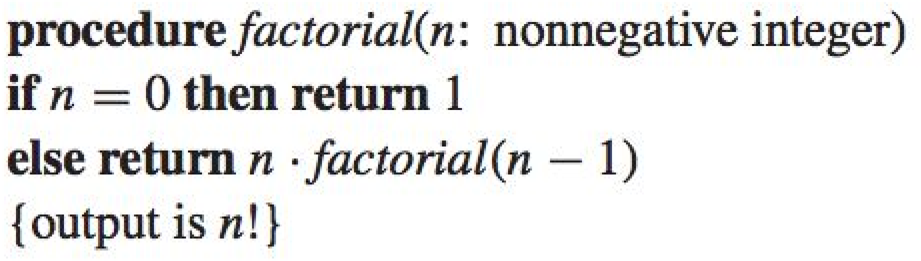
Base Case:

A **base case** is a case, where the problem can be solved without further recursion.
A recursion can lead to an infinite loop, if the base case is not met in the calls.

Python implements recursion by pushing information on a call stack at each recursive call, so it remembers where it must return and continue later.

Functions

Using Recursion To Calculate Factorials:



```
procedure factorial($n$: nonnegative integer)
if $n = 0$ then return 1
else return $n \cdot factorial(n - 1)$
{output is $n!$}
```

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1

Replacing the calculated values gives us the following expression
4! = 4 * 3 * 2 * 1

Functions

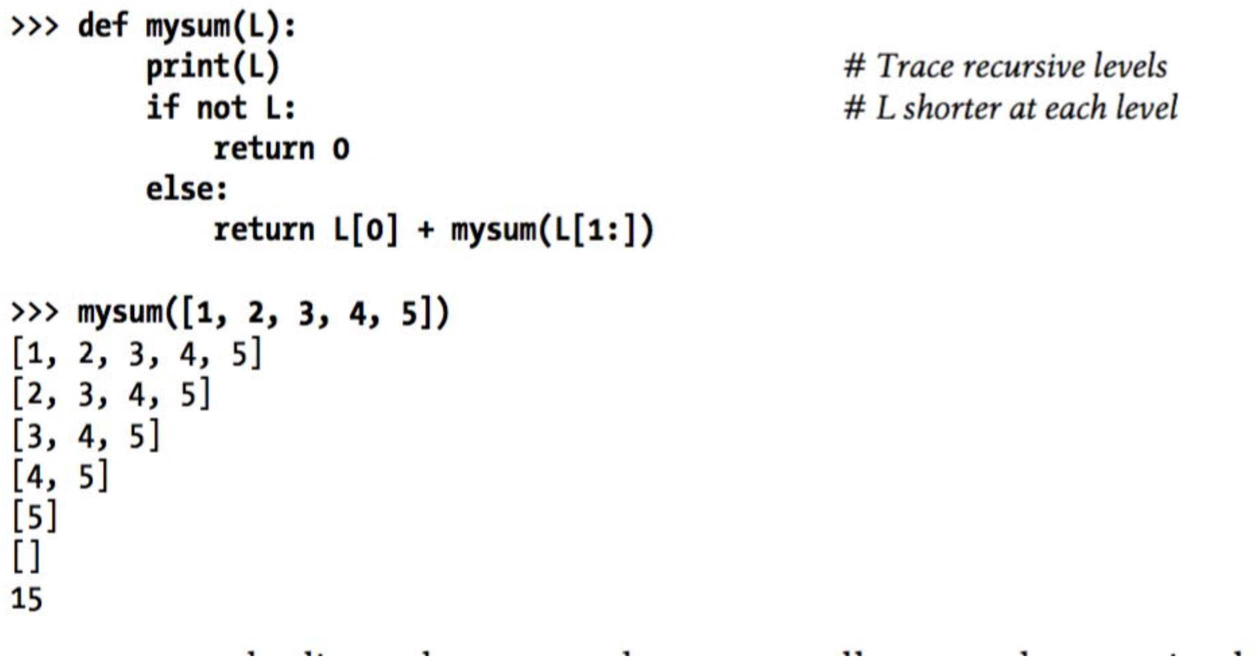
```
def factorial(n):

    print("factorial has been called with n = " + str(n))

    if n == 1:
        return 1

    else:
        res = n * factorial(n-1)

        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)

        return res

print(factorial(5))
```

Output

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1

intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120

120
```

Functions

Recursion to Sum the Elements of a List:

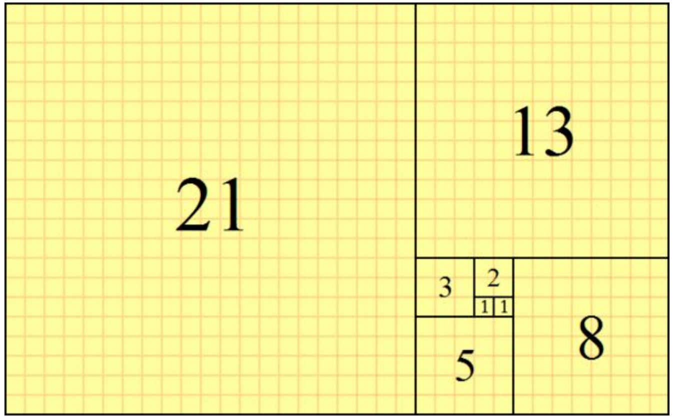


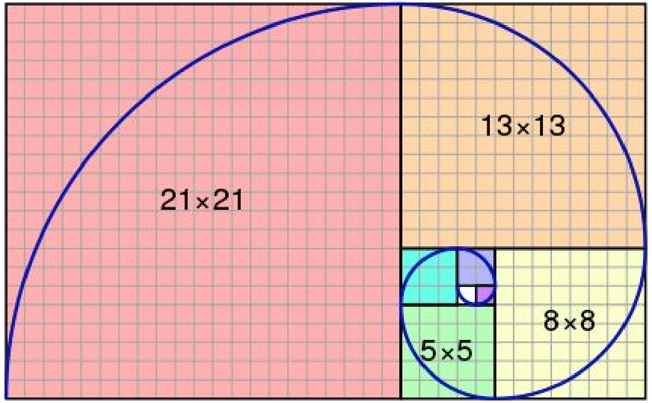
```
>>> def mysum(L):
        print(L)                                    # Trace recursive levels
        if not L:                                   # L shorter at each level
            return 0
        else:
            return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Fibonacci

The **Fibonacci numbers** are the numbers characterized by the fact that every number after the first two is the sum of the two preceding ones. The sequence of Fibonacci numbers appears like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144





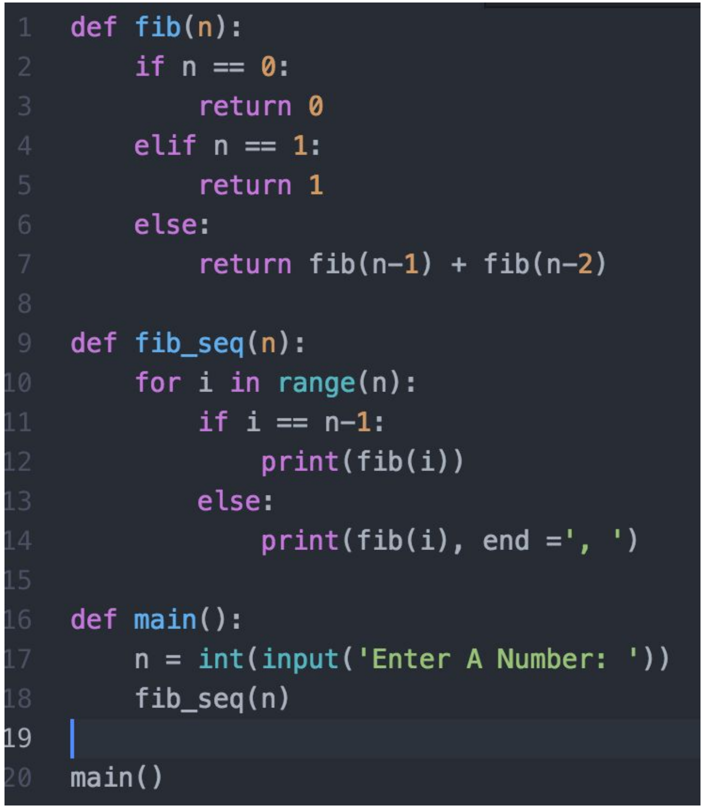
Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Task: Given an integer as input, determine the fibonacci sequence of that length. Use recursion.

i.e.	fib(1)	returns 0
	fib(5)	returns 0,1,1,2,3
	fib(13)	returns 0,1,1,2,3,5,8,13,34,55,89,144

Fibonacci



```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    for i in range(n):
        if i == n-1:
            print(fib(i))
        else:
            print(fib(i), end =', ')

def main():
    n = int(input('Enter A Number: '))
    fib_seq(n)

main()
```

Functions

First Class Functions

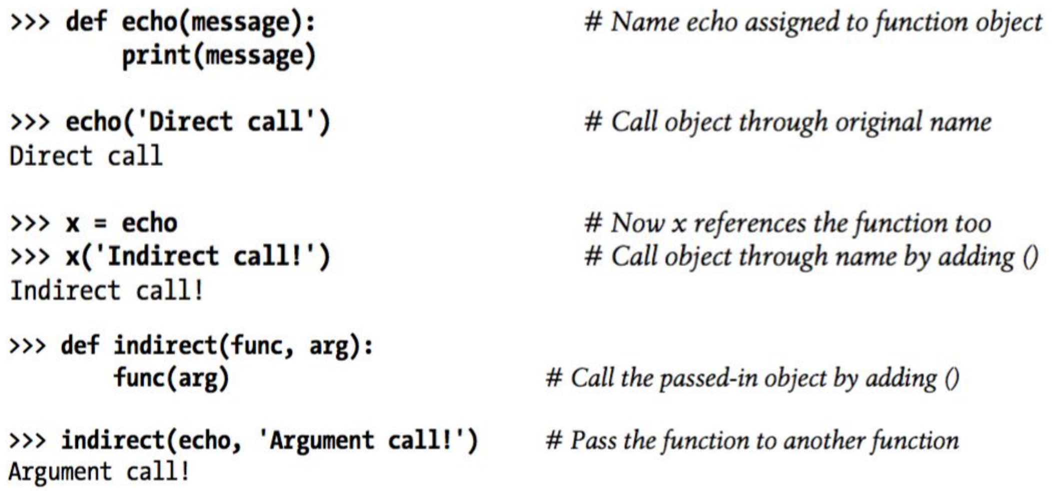
Everything in Python is an object.
This includes functions.
This model is called the *first class object model*.

This means:

- We can assign a function to a variable.
- We can pass a function to another function as an argument.
- We can return a function from another function.
- Etc.

Functions

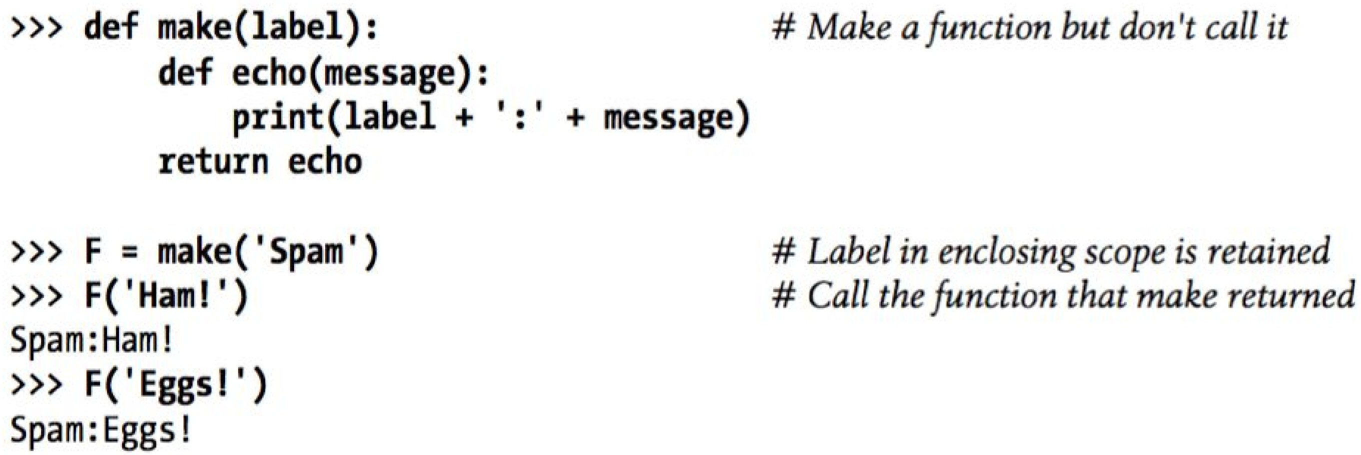
First Class Functions



```
>>> def echo(message):                    # Name echo assigned to function object
        print(message)

>>> echo('Direct call')                   # Call object through original name
Direct call

>>> x = echo                              # Now x references the function too
>>> x('Indirect call!')                   # Call object through name by adding ()
Indirect call!

>>> def indirect(func, arg):
        func(arg)                         # Call the passed-in object by adding ()

>>> indirect(echo, 'Argument call!')      # Pass the function to another function
Argument call!
```

Functions

First Class Functions



```
>>> def make(label):                            # Make a function but don't call it
        def echo(message):
            print(label + ':' + message)
        return echo

>>> F = make('Spam')                            # Label in enclosing scope is retained
>>> F('Ham!')                                   # Call the function that make returned
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

Functions

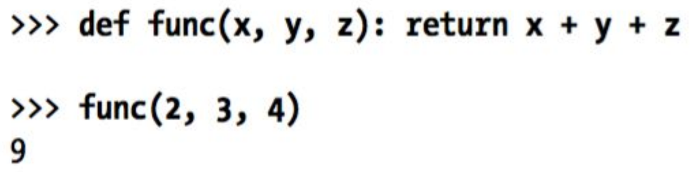
Lambda

Sometimes called **anonymous functions**.

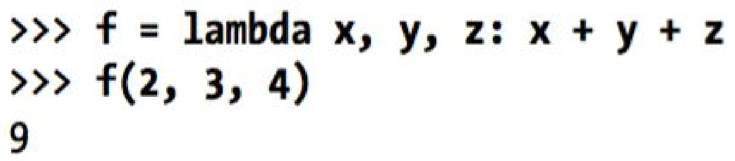
Like def, this expression creates a function, but it returns the function instead of assigning it to a name.

The lambda's general form is the keyword lambda, followed by one or more arguments followed by an expression after a colon:

lambda *argument1, argument2,... argumentN* : *expression using arguments*



```
>>> def func(x, y, z): return x + y + z

>>> func(2, 3, 4)
9
```



```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Functions

Lambda

Lambda is an expression, not a statement

- This means lambda can appear in places a def statement cannot.
  - Inside a list literal, as a function argument,

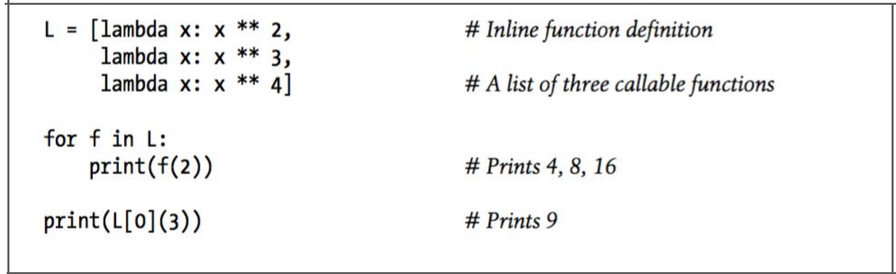
- Lambda is meant for designing simple functions.
- Complex functions should be written with a def statement.

Functions

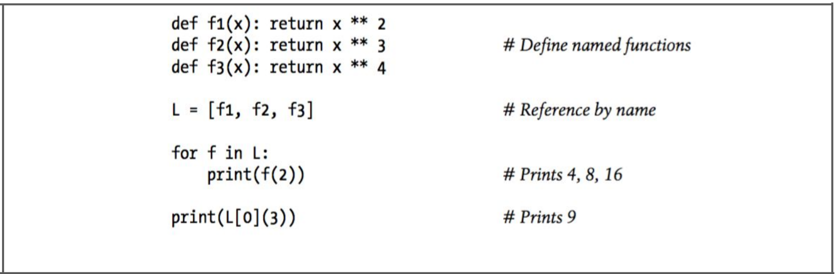
Lambda

Lambda can be used for these reasons but are entirely optional:

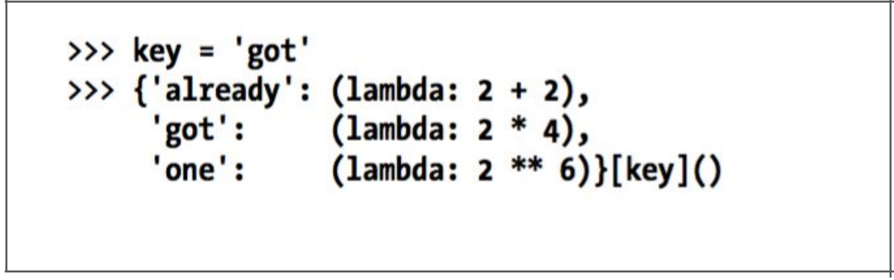
- Write a simple function on a single line
- Jump tables



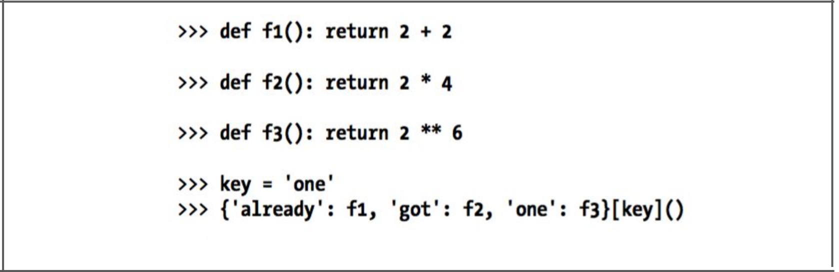
```
L = [lambda x: x ** 2,                  # Inline function definition
     lambda x: x ** 3,
     lambda x: x ** 4]                  # A list of three callable functions

for f in L:
    print(f(2))                         # Prints 4, 8, 16

print(L[0](3))                          # Prints 9
```



```
def f1(x): return x ** 2
def f2(x): return x ** 3                # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]                        # Reference by name

for f in L:
    print(f(2))                         # Prints 4, 8, 16

print(L[0](3))                          # Prints 9
```



```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
     'got':     (lambda: 2 * 4),
     'one':     (lambda: 2 ** 6)}[key]()
```



```
>>> def f1(): return 2 + 2

>>> def f2(): return 2 * 4

>>> def f3(): return 2 ** 6

>>> key = 'one'
>>> {'already': f1, 'got': f2, 'one': f3}[key]()
```

Functions

Generator Functions

Generators produce results only when needed.
This saves memory space and only computes when requested.

Coded like normal **def statements** but use the **yield statement** instead of **return statement**.
Generators implement the **iteration protocol**.

Functions

Generator Functions

Yield Statement

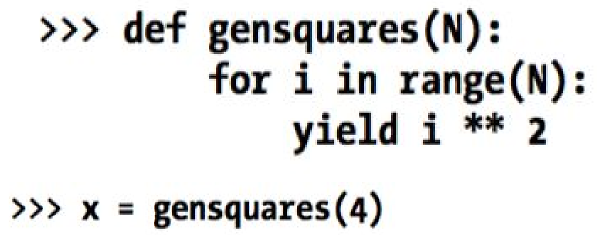
- Yield suspends and saves the state of a function so that the function can be resumed at a later time.

- It also sends one value back to the caller.

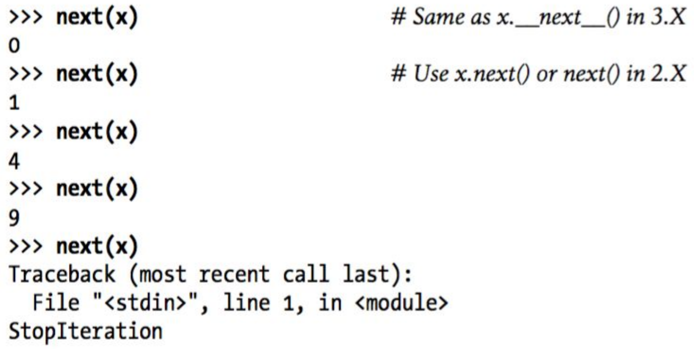
- The yield statement causes the function to adopt the iteration protocol.
  - So the next() method resumes the function.
  - And we can use a for loop to iterate over the generator.

Functions

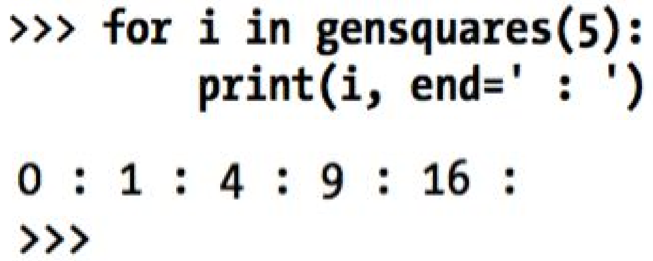
Generator Functions



```
>>> def gensquares(N):
        for i in range(N):
            yield i ** 2

>>> x = gensquares(4)
```



```
>>> next(x)                    # Same as x.__next__() in 3.X
0
>>> next(x)                    # Use x.next() or next() in 2.X
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```



```
>>> for i in gensquares(5):
        print(i, end=' : ')

0 : 1 : 4 : 9 : 16 :
>>>
```

Functions

Decorator Functions

A decorator augments a function's definition by wrapping it in an extra layer of logic.

Syntactically:

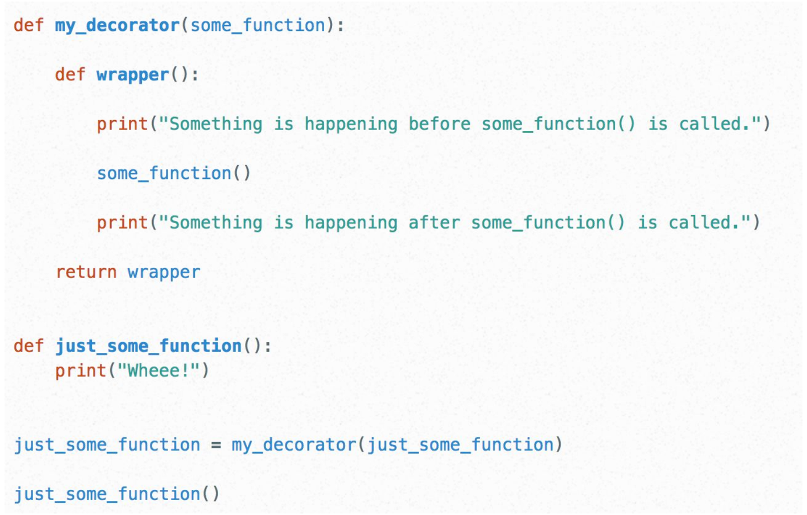
- A function decorator is a sort of runtime declaration about the function that follows.

- A function decorator is coded on a line by itself just before the def statement that defines a function or method.

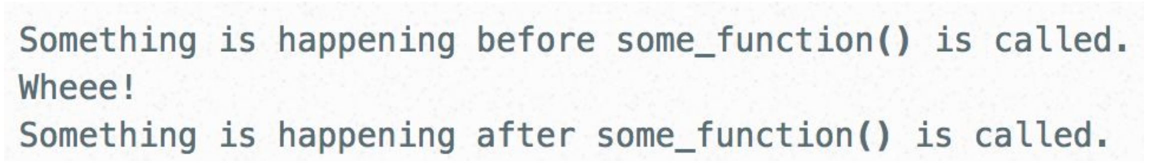
- It consists of the @ symbol, followed by what we call a **metafunction** - a function that manages another function.

Functions

Decorator Functions



```python
def my_decorator(some_function):

    def wrapper():

        print("Something is happening before some_function() is called.")

        some_function()

        print("Something is happening after some_function() is called.")

    return wrapper


def just_some_function():
    print("Wheee!")


just_some_function = my_decorator(just_some_function)

just_some_function()
```



```
Something is happening before some_function() is called.
Wheee!
Something is happening after some_function() is called.
```

Functions

Decorator Functions

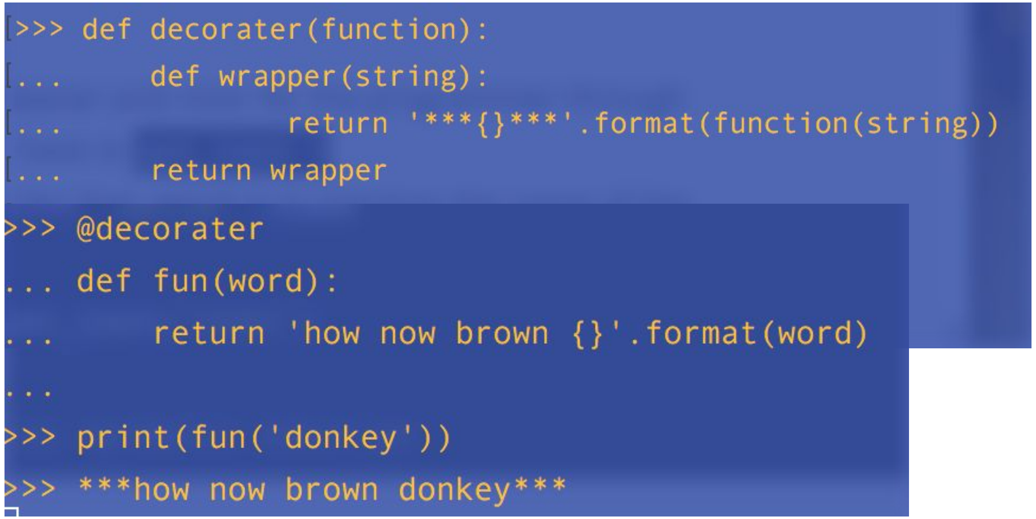
```
@decorator              # Function decoration syntax
    def method():
```

Is equivalent to

```
def method()
method = decorator(method)
```

Functions

Decorator Functions

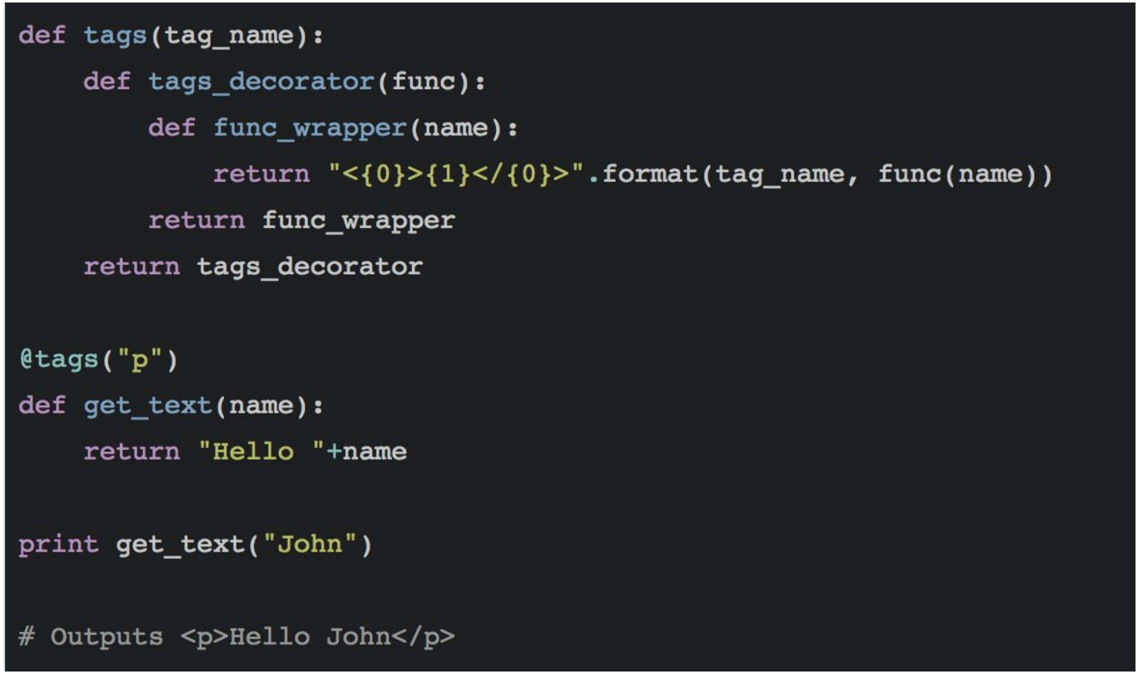


```
[>>> def decorater(function):
[...     def wrapper(string):
[...             return '***{}***'.format(function(string))
[...     return wrapper
```

```
>>> @decorater
... def fun(word):
...     return 'how now brown {}'.format(word)
...
>>> print(fun('donkey'))
>>> ***how now brown donkey***
```

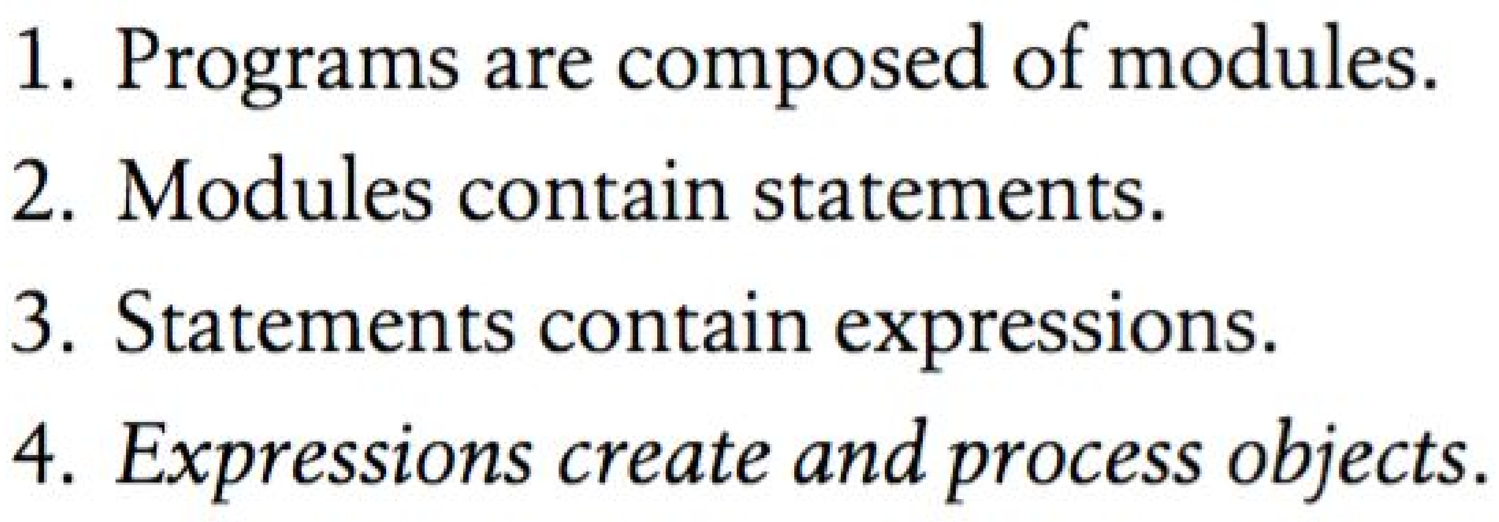
Functions

Decorator Functions With Arguments



```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator

@tags("p")
def get_text(name):
    return "Hello "+name

print get_text("John")

# Outputs <p>Hello John</p>
```

Python Conceptual Hierarchy



1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Import Statement

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.

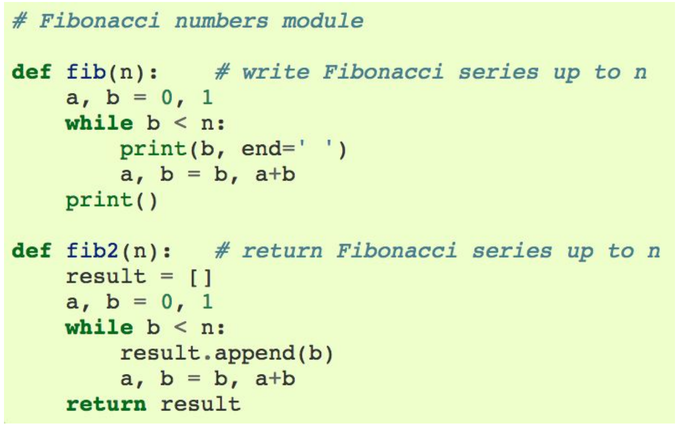
- Such a file is called a ***module***; definitions from a module can be ***imported*** into other modules.

- A module can contain executable statements as well as function definitions.
  - These statements are intended to initialize the module. T
  - They are executed only the *first* time the module name is encountered in an import statement.

- It is customary but not required to place all import statements at the beginning of a module (or script, for that matter).

Import Statement

“fibo.py”



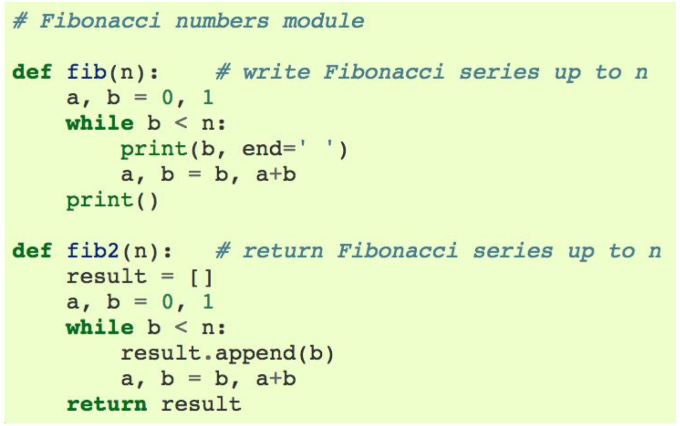
```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Using the module name you can now access the functions:

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Import Statement

“fibo.py”



```python
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Here is a variant of the import statement that imports names from a module directly. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken (so in the example, fibo is not defined).

Import Statement

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_).
In most cases Python programmers do not use this underscore (_) facility.

In general the practice of importing * from a module or package is frowned upon.

Import Statement

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`.
That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

Import Statement

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

Without arguments, dir() lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Built-In Libraries - datetime

A datetime object is a single object containing all the information from a date object and a time object.

Constructor:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

The year, month and day arguments are required.

classmethod datetime.**now**(*tz=None*)

Return the current local date and time.

Instance methods:

datetime.**date**()

Return date object with same year, month and day.

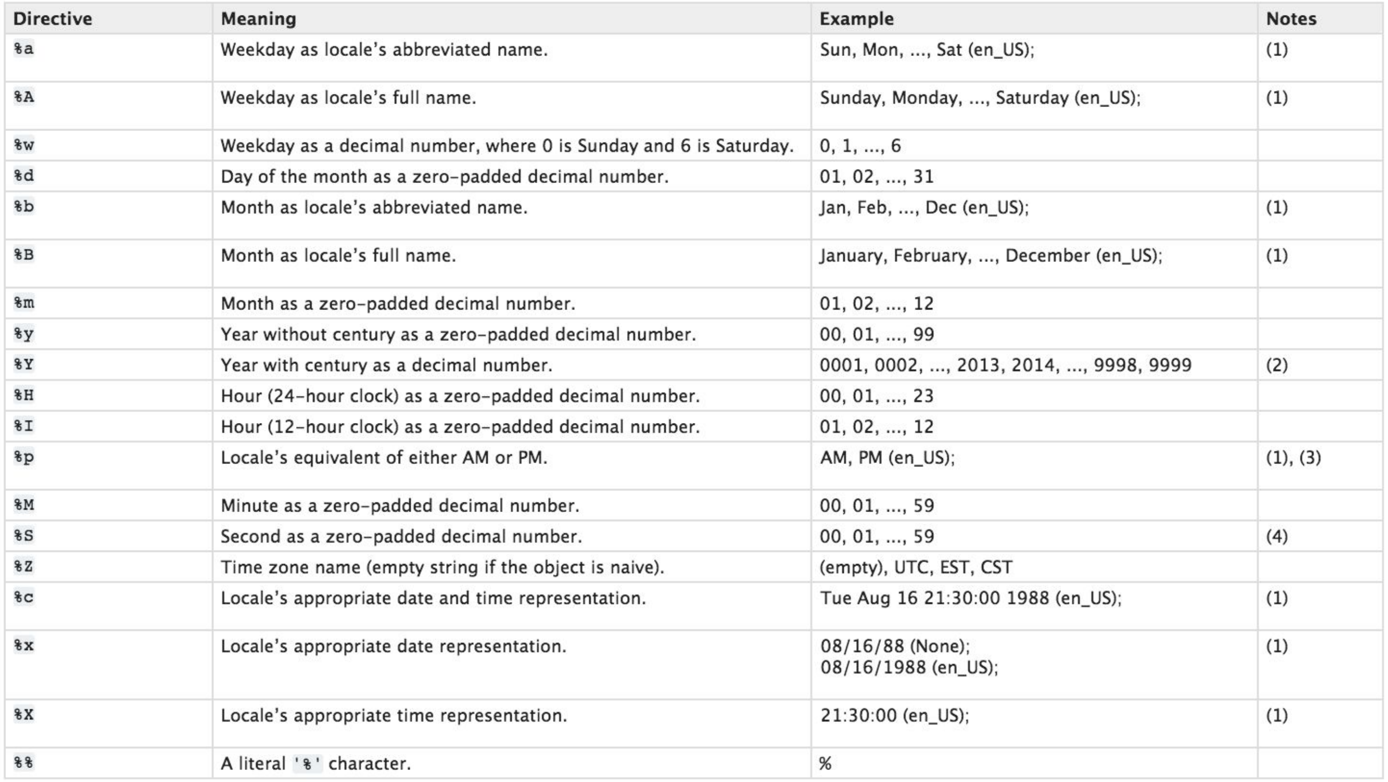
datetime.**time**()

Return time object with same hour, minute, second, microsecond and fold.

datetime.**strftime**(*format*)

Return a string representing the date and time

Built-In Libraries - datetime



Directive	Meaning	Example	Notes
`%a`	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US);	(1)
`%A`	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US);	(1)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6	
`%d`	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	
`%b`	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US);	(1)
`%B`	Month as locale's full name.	January, February, ..., December (en_US);	(1)
`%m`	Month as a zero-padded decimal number.	01, 02, ..., 12	
`%y`	Year without century as a zero-padded decimal number.	00, 01, ..., 99	
`%Y`	Year with century as a decimal number.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23	
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12	
`%p`	Locale's equivalent of either AM or PM.	AM, PM (en_US);	(1), (3)
`%M`	Minute as a zero-padded decimal number.	00, 01, ..., 59	
`%S`	Second as a zero-padded decimal number.	00, 01, ..., 59	(4)
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, EST, CST	
`%c`	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US);	(1)

`%x`	Locale's appropriate date representation.	08/16/88 (None);
		08/16/1988 (en_US);
			(1)

`%X`	Locale's appropriate time representation.	21:30:00 (en_US);	(1)
`%%`	A literal `'%'` character.	%	

Built-In Libraries - datetime

```
>>> dt = datetime.datetime(2006, 11, 21, 16, 30)

>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'

 >>> dt.strftime("%x %X")

>>> datetime.datetime.now()
```

Built-In Libraries - random

random.**randrange**(*start*, *stop*[, *step*])

Return a randomly selected element from range(start, stop, step).

random.**choice**(*seq*)

Return a random element from the non-empty sequence *seq*.

random.**choices**(*population*, *weights=None*, *k=1*)

Return a *k* sized list of elements chosen from the *population* with replacement.
If a *weights* sequence is specified, selections are made according to the relative weights.

random.**shuffle**(*x*)

Shuffle the sequence *x* in place.
To shuffle an immutable sequence and return a new shuffled list, use sample(x, k=len(x)) instead.

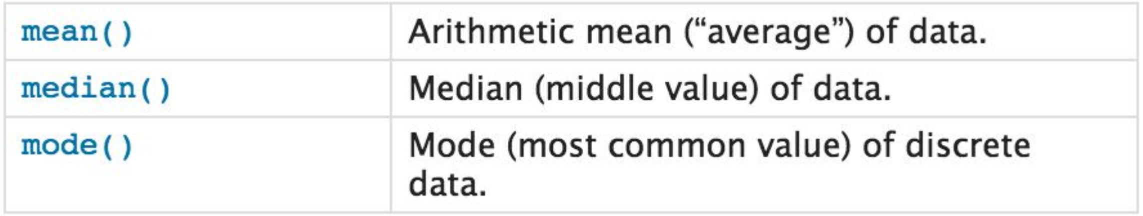
random.**sample**(*population*, *k*)

Return a *k* length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

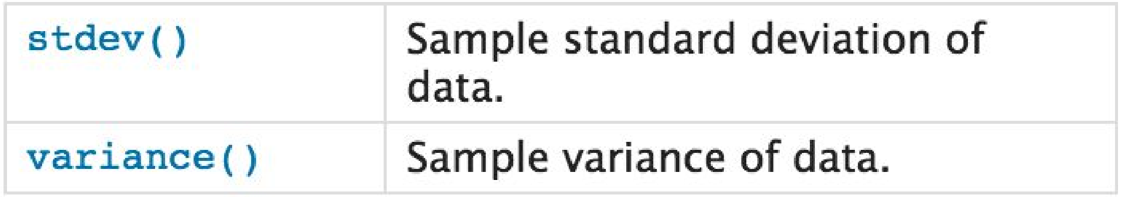
Built-In Libraries - random

```
>>> randrange(10)                              # Integer from 0 to 9 inclusive
7

>>> choice(['win', 'lose', 'draw'])            # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                              # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)          # Four samples without replacement
[40, 10, 50, 30]
```

Built-In Libraries - statistics



`mean()`	Arithmetic mean (“average”) of data.
`median()`	Median (middle value) of data.
`mode()`	Mode (most common value) of discrete data.



`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

Built-In Libraries - csv

The CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases.

csv.**reader**(*csvfile*)
  Return a reader object which will iterate over lines in the given *csvfile*.

csv.**writer**(*csvfile*)
  Return a writer object responsible for converting the user's data into delimited strings on the given file-like object.

class csv.**DictReader**(*f, fieldnames*)
  Create an object that operates like a regular reader but maps the information in each row to an OrderedDict whose keys are given by the optional *fieldnames* parameter.

class csv.**DictWriter**(*f, fieldnames*)
  Create an object which operates like a regular writer but maps dictionaries onto output rows.The *fieldnames* parameter is a sequence of keys that identify the order in which values in the dictionary passed to the `writerow()` method are written to file *f*.

Built-In Libraries - csv

```
import csv

with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])

with open('eggs.csv', newline='') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
        for row in spamreader:
            print(', '.join(row))
```

Built-In Libraries - csv

```python
import csv

with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Eric', 'last_name': 'Idle'})
    writer.writerow({'first_name': 'John', 'last_name': 'Cleese'})
```

```python
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
```

Object Oriented Programming

Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data (functional programming). The programming challenge was seen as how to write the logic, not how to define the data.

Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.

Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on a computer desktop (such as buttons and scrollbars).

OOP is a programming paradigm based on the concept of "**objects**", which may contain:

- data, often known as **attributes**
- and code, often known as **method**

Object Oriented Programming

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Method :** A special kind of function that is defined in a class definition.
- **Instantiation:** The creation of an instance of a class.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Object Oriented Programming - Class Definition

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions must be executed before they have any effect.

In practice, the statements inside a class definition will usually be function definitions.

Object Oriented Programming - Attribute References

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`.

So, if the class definition looked like this:

```python
class MyClass:
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively.

Class attributes can also be assigned to, so you can change the value of `MyClass.i`

Object Oriented Programming - Class Instantiation

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

Object Oriented Programming - Class Instantiation

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

```
def __init__(self):
    self.data = []
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Object Oriented Programming - Instance Objects

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Object Oriented Programming - Instance Objects

The other kind of instance attribute reference is a *method*.

A method is a function that “belongs to” an object.

Usually, a method is called right after it is bound:

```
x.f()
```

In the MyClass example, this will return the string `'hello world'`.

However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `hello world` until the end of time.

Object Oriented Programming - Instance Objects

You may have noticed that `x.f()` was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? .

The special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`.

Object Oriented Programming

```
class Dog:

    kind = 'canine'                    # class variable shared by all instances

    def __init__(self, name):
        self.name = name               # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                             # shared by all dogs
'canine'
>>> e.kind                             # shared by all dogs
'canine'
>>> d.name                              # unique to d
'Fido'
>>> e.name                             # unique to e
```

Object Oriented Programming

```
class Dog:

    tricks = []                         # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                            # unexpectedly shared by all dogs
['roll over', 'play dead']
```

The *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances.

Object Oriented Programming

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Object Oriented Programming - Inheritance

The syntax for inheritance definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    <statement-N>
```

When the class object is constructed, the base class is remembered.
This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class.
There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class.

Derived classes may override methods of their base classes.
There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`.

Python has two built-in functions that work with inheritance:

- Use isinstance() to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is int or some class derived from int.
- Use issubclass() to check class inheritance: `issubclass(bool, int)` is `True` since bool is a subclass of int. However, `issubclass(float,int)` is `False` since float is not a subclass of int.

Object Oriented Programming

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr
```

```
class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Object Oriented Programming

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```