

CHAPTER 1. YOUR FIRST PYTHON PROGRAM

“ Don’t bury your burden in saintly silence. You have a problem? Great. Rejoice, dive in, and investigate. ”

— Ven. Henepola Gunaratana

DIVING IN

Convention dictates that I should bore you with the fundamental building blocks of programming, so we can slowly work up to building something useful. Let’s skip all that. Here is a complete, working Python program. It probably makes absolutely no sense to you. Don’t worry about that, because you’re going to dissect it line by line. But read through it first and see what, if anything, you can make of it.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    ...

    if size < 0:
        raise ValueError('number must be non-negative')

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
```

```

    size /= multiple

    if size < multiple:
        return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(100000000000, False))
    print(approximate_size(1000000000000))

```

Now let's run this program on the command line. On Windows, it will look something like this:

```

c:\home\diveintopython3\examples> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB

```

On Mac OS X or Linux, it would look something like this:

```

you@localhost:~/diveintopython3/examples$ python3 humansize.py
1.0 TB
931.3 GiB

```

What just happened? You executed your first Python program. You called the Python interpreter on the command line, and you passed the name of the script you wanted Python to execute. The script defines a single function, the `approximate_size()` function, which takes an exact file size in bytes and calculates a “pretty” (but approximate) size. (You’ve probably seen this in Windows Explorer, or the Mac OS X Finder, or Nautilus or Dolphin or Thunar on Linux. If you display a folder of documents as a multi-column list, it will display a table with the document icon, the document name, the size, type, last-modified date, and so on. If the folder contains a 1093-byte file named `TODO`, your file manager won’t display `TODO 1093 bytes`; it’ll say something like `TODO 1 KB` instead. That’s what the `approximate_size()` function does.)

Look at the bottom of the script, and you’ll see two calls to `print(approximate_size(arguments))`. These are function calls — first calling the `approximate_size()` function and passing a number of arguments, then taking the return value and passing it straight on to the `print()` function. The `print()` function is built-in; you’ll

never see an explicit declaration of it. You can just use it, anytime, anywhere. (There are lots of built-in functions, and lots more functions that are separated into *modules*. Patience, grasshopper.)

So why does running the script on the command line give you the same output every time? We'll get to that. First, let's look at that `approximate_size()` function.

*
**

1.2. DECLARING FUNCTIONS

Python has functions like most other languages, but it does not have separate header files like `c++` or interface/implementation sections like Pascal. When you need a function, just declare it, like this:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

The keyword `def` starts the function declaration, followed by the function name, followed by the arguments in parentheses. Multiple arguments are separated with commas.

Also note that the function doesn't define a return datatype. Python functions do not specify the datatype of their return value; they don't even specify whether or not they return a value. (In fact, every Python function returns a value; if the function ever executes a `return` statement, it will return that value, otherwise it will return `None`, the Python null value.)

*When you
need a
function, just
declare it.*



In some languages, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

The `approximate_size()` function takes the two arguments — `size` and `a_kilobyte_is_1024_bytes` — but neither argument specifies a datatype. In Python, variables are never explicitly typed. Python figures out what type a variable is and keeps track of it internally.



In Java and other statically-typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.

1.2.1. OPTIONAL AND NAMED ARGUMENTS

Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Furthermore, arguments can be specified in any order by using named arguments.

Let's take another look at that `approximate_size()` function declaration:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

The second argument, `a_kilobyte_is_1024_bytes`, specifies a default value of `True`. This means the argument is *optional*; you can call the function without it, and Python will act as if you had called it with `True` as a second parameter.

Now look at the bottom of the script:

```
if __name__ == '__main__':  
    print(approximate_size(1000000000000, False)) ①  
    print(approximate_size(1000000000000))        ②
```

- ① This calls the `approximate_size()` function with two arguments. Within the `approximate_size()` function, `a_kilobyte_is_1024_bytes` will be `False`, since you explicitly passed `False` as the second argument.
- ② This calls the `approximate_size()` function with only one argument. But that's OK, because the second argument is optional! Since the caller doesn't specify, the second argument defaults to `True`, as defined by

the function declaration.

You can also pass values into a function by name.

```
>>> from humansize import approximate_size
>>> approximate_size(4000, a_kilobyte_is_1024_bytes=False) ①
'4.0 KB'
>>> approximate_size(size=4000, a_kilobyte_is_1024_bytes=False) ②
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, size=4000) ③
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, 4000) ④
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> approximate_size(size=4000, False) ⑤
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

- ① This calls the `approximate_size()` function with 4000 for the first argument (`size`) and `False` for the argument named `a_kilobyte_is_1024_bytes`. (That happens to be the second argument, but doesn't matter, as you'll see in a minute.)
- ② This calls the `approximate_size()` function with 4000 for the argument named `size` and `False` for the argument named `a_kilobyte_is_1024_bytes`. (These named arguments happen to be in the same order as the arguments are listed in the function declaration, but that doesn't matter either.)
- ③ This calls the `approximate_size()` function with `False` for the argument named `a_kilobyte_is_1024_bytes` and 4000 for the argument named `size`. (See? I told you the order didn't matter.)
- ④ This call fails, because you have a named argument followed by an unnamed (positional) argument, and that never works. Reading the argument list from left to right, once you have a single named argument, the rest of the arguments must also be named.
- ⑤ This call fails too, for the same reason as the previous call. Is that surprising? After all, you passed 4000 for the argument named `size`, then "obviously" that `False` value was meant for the `a_kilobyte_is_1024_bytes` argument. But Python doesn't work that way. As soon as you have a named argument, all arguments to the right of that need to be named arguments, too.



1.3. WRITING READABLE CODE

I won't bore you with a long finger-wagging speech about the importance of documenting your code. Just know that code is written once but read many times, and the most important audience for your code is yourself, six months after writing it (i.e. after you've forgotten everything but need to fix something). Python makes it easy to write readable code, so take advantage of it. You'll thank me in six months.

1.3.1. DOCUMENTATION STRINGS

You can document a Python function by giving it a documentation string (docstring for short). In this program, the `approximate_size()` function has a docstring:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):  
    '''Convert a file size to human-readable form.  
  
    Keyword arguments:  
    size -- file size in bytes  
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024  
                               if False, use multiples of 1000  
  
    Returns: string  
  
    ...
```

Triple quotes signify a multi-line string. Everything between the start and end quotes is part of a single string, including carriage returns, leading white space, and other quote characters. You can use them anywhere, but you'll see them most often used when defining a docstring.

*Every
function*

☞ Triple quotes are also an easy way to define a string with both single and double quotes, like `qq/.../` in Perl 5.

*deserves a
decent
docstring.*

Everything between the triple quotes is the function's docstring, which documents what the function does. A docstring, if it exists, must be the first thing defined in a function (that is, on the next line after the function declaration). You don't technically need to give your function a docstring, but you always should. I know you've heard this in every programming class you've ever taken, but Python gives you an added incentive: the docstring is available at runtime as an attribute of the function.

☞ Many Python IDEs use the docstring to provide context-sensitive documentation, so that when you type a function name, its docstring appears as a tooltip. This can be incredibly helpful, but it's only as good as the docstrings you write.

*
**

1.4. THE `import` SEARCH PATH

Before this goes any further, I want to briefly mention the library search path. Python looks in several places when you try to import a module. Specifically, it looks in all the directories defined in `sys.path`. This is just a list, and you can easily view it or modify it with standard list methods. (You'll learn more about lists in [Native Datatypes](#).)

```
>>> import sys                                ①
>>> sys.path                                  ②
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
```

```

'/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
'/usr/lib/python3.1/lib-dynload',
'/usr/lib/python3.1/dist-packages',
'/usr/local/lib/python3.1/dist-packages']

>>> sys ③

<module 'sys' (built-in)>

>>> sys.path.insert(0, '/home/mark/diveintopython3/examples') ④

>>> sys.path ⑤

['/home/mark/diveintopython3/examples',
'',
'/usr/lib/python31.zip',
'/usr/lib/python3.1',
'/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
'/usr/lib/python3.1/lib-dynload',
'/usr/lib/python3.1/dist-packages',
'/usr/local/lib/python3.1/dist-packages']

```

- ① Importing the `sys` module makes all of its functions and attributes available.
- ② `sys.path` is a list of directory names that constitute the current search path. (Yours will look different, depending on your operating system, what version of Python you're running, and where it was originally installed.) Python will look through these directories (in this order) for a `.py` file whose name matches what you're trying to import.
- ③ Actually, I lied; the truth is more complicated than that, because not all modules are stored as `.py` files. Some are *built-in modules*; they are actually baked right into Python itself. Built-in modules behave just like regular modules, but their Python source code is not available, because they are not written in Python! (Like Python itself, these built-in modules are written in C.)
- ④ You can add a new directory to Python's search path at runtime by adding the directory name to `sys.path`, and then Python will look in that directory as well, whenever you try to import a module. The effect lasts as long as Python is running.
- ⑤ By using `sys.path.insert(0, new_path)`, you inserted a new directory as the first item of the `sys.path` list, and therefore at the beginning of Python's search path. This is almost always what you want. In case of naming conflicts (for example, if Python ships with version 2 of a particular library but you want to use version 3), this ensures that your modules will be found and used instead of the modules that came with Python.



1.5. EVERYTHING IS AN OBJECT

In case you missed it, I just said that Python functions have attributes, and that those attributes are available at runtime. A function, like everything else in Python, is an object.

Run the interactive Python shell and follow along:

```
>>> import humansize ①
>>> print(humansize.approximate_size(4096, True)) ②
4.0 KiB
>>> print(humansize.approximate_size.__doc__) ③
```

Convert a file size to human-readable form.


Keyword arguments:

size -- file size in bytes

a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
if False, use multiples of 1000

Returns: string

- ① The first line imports the `humansize` program as a module — a chunk of code that you can use interactively, or from a larger Python program. Once you import a module, you can reference any of its public functions, classes, or attributes. Modules can do this to access functionality in other modules, and you can do it in the Python interactive shell too. This is an important concept, and you'll see a lot more of it throughout this book.
- ② When you want to use functions defined in imported modules, you need to include the module name. So you can't just say `approximate_size`; it must be `humansize.approximate_size`. If you've used classes in Java, this should feel vaguely familiar.
- ③ Instead of calling the function as you would expect to, you asked for one of the function's attributes, `__doc__`.

 `import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with `module.function`; once you `require` a Perl module, you access its functions with `module::function`.

1.5.1. WHAT'S AN OBJECT?

Everything in Python is an object, and everything can have attributes and methods. All functions have a built-in attribute `__doc__`, which returns the docstring defined in the function's source code. The `sys` module is an object which has (among other things) an attribute called `path`. And so forth.

Still, this doesn't answer the more fundamental question: what is an object? Different programming languages define "object" in different ways. In some, it means that *all* objects *must* have attributes and methods; in others, it means that all objects are subclassable. In Python, the definition is looser. Some objects have neither attributes nor methods, *but they could*. Not all objects are subclassable. But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function.

You may have heard the term "first-class object" in other programming contexts. In Python, functions are *first-class objects*. You can pass a function as an argument to another function. Modules are *first-class objects*. You can pass an entire module as an argument to a function. Classes are first-class objects, and individual instances of a class are also first-class objects.

This is important, so I'm going to repeat it in case you missed it the first few times: *everything in Python is an object*. Strings are objects. Lists are objects. Functions are objects. Classes are objects. Class instances are objects. Even modules are objects.

*
**

1.6. INDENTING CODE

Python functions have no explicit `begin` or `end`, and no curly braces to mark where the function code starts and stops. The only delimiter is a colon (`:`) and the indentation of the code itself.

```

def approximate_size(size, a_kilobyte_is_1024_bytes=True): ①
    if size < 0: ②
        raise ValueError('number must be non-negative') ③
    ④
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]: ⑤
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')

```

- ① Code blocks are defined by their indentation. By “code block,” I mean functions, if statements, for loops, while loops, and so forth. Indenting starts a block and unindenting ends it. There are no explicit braces, brackets, or keywords. This means that whitespace is significant, and must be consistent. In this example, the function code is indented four spaces. It doesn’t need to be four spaces, it just needs to be consistent. The first line that is not indented marks the end of the function.
- ② In Python, an if statement is followed by a code block. If the if expression evaluates to true, the indented block is executed, otherwise it falls to the else block (if any). Note the lack of parentheses around the expression.
- ③ This line is inside the if code block. This raise statement will raise an exception (of type ValueError), but only if `size < 0`.
- ④ This is *not* the end of the function. Completely blank lines don’t count. They can make the code more readable, but they don’t count as code block delimiters. The function continues on the next line.
- ⑤ The for loop also marks the start of a code block. Code blocks can contain multiple lines, as long as they are all indented the same amount. This for loop has three lines of code in it. There is no other special syntax for multi-line code blocks. Just indent and get on with your life.

After some initial protests and several snide analogies to Fortran, you will make peace with this and start seeing its benefits. One major benefit is that all Python programs look similar, since indentation is a language requirement and not a matter of style. This makes it easier to read and understand other people’s Python code.

👉 Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.

*
**

1.7. EXCEPTIONS

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances. You'll see them repeatedly throughout this book.


What is an exception? Usually it's an error, an indication that something went wrong. (Not all exceptions are errors, but never mind that for now.) Some programming languages encourage the use of error return codes, which you *check*. Python encourages the use of exceptions, which you *handle*.

When an error occurs in the Python Shell, it prints out some details about the exception and how it happened, and that's that. This is called an *unhandled* exception. When the exception was raised, there was no code to explicitly notice it and deal with it, so it bubbled its way back up to the top level of the Python Shell, which spits out some debugging information and calls it a day. In the shell, that's no big deal, but if that happened while your actual Python program was running, the entire program would come to a screeching halt if nothing handles the exception. Maybe that's what you want, maybe it isn't.

👉 Unlike Java, Python functions don't declare which exceptions they might raise. It's up to you to determine what possible exceptions you need to catch.

An exception doesn't need to result in a complete program crash, though. Exceptions can be *handled*. Sometimes an exception is really because you have a bug in your code (like accessing a variable that doesn't exist), but sometimes an exception is something you can anticipate. If you're opening a file, it might not exist. If you're importing a module, it might not be installed. If you're connecting to a database, it might be


unavailable, or you might not have the correct security credentials to access it. If you know a line of code may raise an exception, you should handle the exception using a `try...except` block.

 Python uses `try...except` blocks to handle exceptions, and the `raise` statement to generate them. Java and C++ use `try...catch` blocks to handle exceptions, and the `throw` statement to generate them.

The `approximate_size()` function raises exceptions in two different cases: if the given size is larger than the function is designed to handle, or if it's less than zero.

```
if size < 0:
    raise ValueError('number must be non-negative')
```

The syntax for raising an exception is simple enough. Use the `raise` statement, followed by the exception name, and an optional human-readable string for debugging purposes. The syntax is reminiscent of calling a function. (In reality, exceptions are implemented as classes, and this `raise` statement is actually creating an instance of the `ValueError` class and passing the string 'number must be non-negative' to its initialization method. But we're getting ahead of ourselves!)

 You don't need to handle an exception in the function that raises it. If one function doesn't handle it, the exception is passed to the calling function, then that function's calling function, and so on "up the stack." If the exception is never handled, your program will crash, Python will print a "traceback" to standard error, and that's the end of that. Again, maybe that's what you want; it depends on what your program does.

1.7.1. CATCHING IMPORT ERRORS

One of Python's built-in exceptions is `ImportError`, which is raised when you try to import a module and fail. This can happen for a variety of reasons, but the simplest case is when the module doesn't exist in your import search path. You can use this to include optional features in your program. For example, the `chardet`

library provides character encoding auto-detection. Perhaps your program wants to use this library *if it exists*, but continue gracefully if the user hasn't installed it. You can do this with a `try...except` block.

```
try:
    import chardet
except ImportError:
    chardet = None
```

Later, you can check for the presence of the `chardet` module with a simple `if` statement:

```
if chardet:
    # do something
else:
    # continue anyway
```

Another common use of the `ImportError` exception is when two modules implement a common API, but one is more desirable than the other. (Maybe it's faster, or it uses less memory.) You can try to import one module but fall back to a different module if the first import fails. For example, the XML chapter talks about two modules that implement a common API, called the `ElementTree` API. The first, `lxml`, is a third-party module that you need to download and install yourself. The second, `xml.etree.ElementTree`, is slower but is part of the Python 3 standard library.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

By the end of this `try...except` block, you have imported *some* module and named it `etree`. Since both modules implement a common API, the rest of your code doesn't need to keep checking which module got imported. And since the module that *did* get imported is always called `etree`, the rest of your code doesn't need to be littered with `if` statements to call differently-named modules.



1.8. UNBOUND VARIABLES

Take another look at this line of code from the `approximate_size()` function:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

You never declare the variable `multiple`, you just assign a value to it. That's OK, because Python lets you do that. What Python will *not* let you do is reference a variable that has never been assigned a value. Trying to do so will raise a `NameError` exception.

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 1
>>> x
1
```

You will thank Python for this one day.



1.9. EVERYTHING IS CASE-SENSITIVE

All names in Python are case-sensitive: variable names, function names, class names, module names, exception names. If you can get it, set it, call it, construct it, import it, or raise it, it's case-sensitive.

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'AN_INTEGER' is not defined
```

```
>>> An_Integer
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
NameError: name 'An_Integer' is not defined
```

```
>>> an_inteGer
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
NameError: name 'an_inteGer' is not defined
```

And so on.

*
**

1.10. RUNNING SCRIPTS

Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them, by including a special block of code that executes when you run the Python file on the command line. Take the last few lines of `humansize.py`:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

*Everything in
Python is an
object.*



Like `c`, Python uses `==` for comparison and `=` for assignment. Unlike `c`, Python does not support in-line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

So what makes this `if` statement special? Well, modules are objects, and all modules have a built-in attribute `__name__`. A module's `__name__` depends on how you're using the module. If you `import` the module, then `__name__` is the module's filename, without a directory path or file extension.

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

But you can also run the module directly as a standalone program, in which case `__name__` will be a special default value, `__main__`. Python will evaluate this `if` statement, find a true expression, and execute the `if` code block. In this case, to print two values.

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

And that's your first Python program!

*
**

1.11. FURTHER READING

- [PEP 257: Docstring Conventions](#) explains what distinguishes a good docstring from a great docstring.
- [Python Tutorial: Documentation Strings](#) also touches on the subject.
- [PEP 8: Style Guide for Python Code](#) discusses good indentation style.
- [Python Reference Manual](#) explains what it means to say that everything in Python is an object, because some people are pedants and like to discuss that sort of thing at great length.



© 2001–II Mark Pilgrim