Programming with Python

Stefan Güttel, guettel.com (http://guettel.com)

Contents

Functions

Functions are an extremely important programming concept for structuring your code and avoiding repetitions. And you will use them extensively in the upcoming lab tests and also in the final coursework. So let's dive in!

We have previously seen a code similar to the one below, while discussing various problems arising with floating point operations on a computer:

In [1]:

```
 = int(input("n = "))
0 = 0
1 = 1
hile n > 1:
   nxt = f0 + f1
   f0 = f1
   f1 = nxt
   n -= 1
rint("Fn =", f1)
```

```
n = 7
Fn = 13
```

This code inputs an integer `n` for which it computes and prints the n^{th} Fibonacci number.

Here is how we can analyze it:

1. We first initialize (set to their starting values) variables `f0` and `f1` to 0 and 1 respectively.
2. Then we run a loop as long as `n > 1`. In that loop, we

   2.1. Compute the sum `f0 + f1` and save it for later in the variable `nxt` (short of "next", which we don't use because `next` is a built-in Python function).

   2.2. We assign `f0 = f1` and `f1 = nxt`.
   In other words, the new value of `f0` is equal to the old value of `f1` and the new value is equal to the sum of **old** values of `f0` and `f1`.

   2.3. We reduce `n` by 1.

Notice how `n` is changed exactly once in each pass through the loop and nowhere else. This means that our variable `n` has the following values throughout the loop:
$n \to n-1$ (the 1^{st} pass; drops to $n-1$ in the last step of the loop),
$n-1 \to n-2$ (the 2^{nd} pass; drops to $n-2$ in the last step of the loop),
$n-2 \to n-3$ (the 3^{rd} pass; drops to $n-3$ in the last step of the loop),
...
$n-k+1 \to n-k$ (the k^{th} pass; drops to $n-k$ in the last step of the loop),
...

The loop will stop when the current value of `n` no longer satisfies the condition `n > 1`. Given that our variable drops by 1 in each step, this will happen when it drops to exactly 1. In other words, when variable `n` gets the value `1 = n - k`, which happens at the end step number $k = n - 1$ (where n is the starting value of the variable `n`).

Given that `f0 =` F_0 and `f1 =` F_1 are the zeroth and the first Fibonacci numbers, and all subsequent ones are the sum of the previous two (`nxt = f0 + f1`), we conclude that the above code produces the Fibonacci numbers. How many of them?

Each loop pass produces one Fibinacci number (in the first pass we get F_2, then in the second we get F_3, etc.) and we have just concluded that we do this $k = n - 1$ times. In other words, when we're done, `f1` contains the value F_n.

From a regular code to a function

Now, computing the n^{th} Fibonacci number is sometimes useful and it would be really neat to write it once and then (re)use it throughout the code. This is achieved using **functions**.

Here is the previously displayed code, wrapped in a function:

In [2]:

```
def fibonacci(n):
    """
    Takes one argument `n` and returns the `n`-th Fibonacci number.
    """
    f0 = 0
    f1 = 1
    while n > 1:
        nxt = f0 + f1
        f0 = f1
        f1 = nxt
        n -= 1
    return f1
```

Here are the main elements:

1. Function declaration, consisting of these elements:
   A. the keyword `def`;
   B. the name of the function;
   C. the list of the *formal parameters* that the function takes, enclosed in brackets.
      The brackets have to be there, even if the function takes no arguments.
   D. a colon (as always when we are starting a block).
2. A so called *docstring* -- an explanation what the function does, enclosed in a pair of triple quotation marks. This is a fairly pythonic thing that exists in some form in only a handful of other languages (PERL, for example).
   Recall that we've seen these last week, and we're putting them in the beginning of all programs.
   While technically not mandatory, it is a good practice to thoroughly comment your parts of your code in Python (all the functions, classes, modules, etc.).
3. The body of the function: this is the code that is executed when the function is envoked. It is pretty much the same as the "regular" code, apart from the occasional `return` statement.
4. The `return` statement does two equally important tasks:
   A. It **interrupts the execution** of the function.
   B. It provides the function's return value.

Notice how running the above code does seemingly nothing. It actually creates a function, which we can then call...

Calling a function

This is as straightforward as one would expect:

In [3]:

```
  = int(input("n = "))
  = fibonacci(n)
rint("F(" + str(n) + ") =", f)
```

```
n = 7
F(7) = 13
```

or

In [4]:

```
  = int(input("n = "))
rint("F(" + str(n) + ") =", fibonacci(n))
```

```
n = 7
F(7) = 13
```

In the call `fibonacci(n)`, the variable `n` is called the *actual parameter* or *argument*. Even though it has the same name as the formal parameter, these two are *not* the same variable. Furthermore, the formal and the actual arguments need not have the same name. For example, these calls are all perfectly correct:

In [5]:

```
 = int(input("x = "))
rint("F(" + str(x) + ") =", fibonacci(x))
rint("F(17) =", fibonacci(17))
rint("F(13 + 17) =", fibonacci(13 + 17))
rint("F(x + 7) = F(" + str(x + 7) + ") =", fibonacci(x + 7))
```

```
x = 7
F(7) = 13
F(17) = 1597
F(13 + 17) = 832040
F(x + 7) = F(14) = 377
```

The parameters are (mostly) copies of the arguments

Observe the following simple example:

In [6]:

```
ef inc(x, y):
   x += y
   return x

 = 17
 = inc(a, 2)
rint(a, b)
```

```
17 19
```

As you can see, the value of `a` was not changed, even though the value of `x` was. This is because, when the function was called, `x` took the value of `a` (but it did not become `a`!), so the statement `x += y` changed only that copy, while (the original) `a` remained intact.

To verify that the values of the *local variables* (i.e., those that "belong" to a function and are not visible outside of it) truly vanish when the function finishes, you can always try something like this:

In [7]:

```
 ef test(defX):
    if defX:
        x = 17
    print("x =", x)

 rint("The first call:")
 est(True)
 rint("The second call:")
 est(False)
```

```
The first call:
x = 17
The second call:

---------------------------------------------------------------------
-------
 nboundLocalError                         Traceback (most recent cal
  last)
 ipython-input-7-724421d99ac1> in <module>()
      7 test(True)
      8 print("The second call:")
----> 9 test(False)

 ipython-input-7-724421d99ac1> in test(defX)
      2     if defX:
      3         x = 17
----> 4     print("x =", x)
      5
      6 print("The first call:")

 nboundLocalError: local variable 'x' referenced before assignment
```

Complex data structures

When it comes to lists, sets, and other data structures, things get a bit more complex than "their parameters are copies of the corresponding arguments". We will explain this in detail when we discuss these structures.

More on returning values

Python also has a keyword `yield` that can be used to return list-like-objects (the same thing that `for`-loops like to use) in a special, memory efficient way. More on that when we learn about *generators*.

In many programming languages, functions can return at most one value. In Python, however, a function can return as many values as we like. This will also be addressed later on.

A function that ends without `return` or with just a `return` (no values given to be returned), will return `None`. In other programming languages, these may be called *void-functions* or *procedures*.

If a function has more than one `return` statement (split between by an `if` branch or a similar mechanism), they need not return the same type or even the same number of values. However, be careful how these will be used (anyone calling the function must be aware of how many values are returned, what are their types, and what they represent; otherwise the return values will be useless).

The `return` statement cannot be used outside of a function (it would have no meaning).

Consider the following two examples:

In [8]:

```
 ef max1(x, y):
    """
    Takes comparable values `x` and `y` as arguments.

    Returns the larger of the two, implemented with if-else.
    """
    if x > y:
        return x
    else:
        return y

 ef max2(x, y):
    """
    Takes comparable values `x` and `y` as arguments.

    Returns the larger of the two, implemented with if (no else).
    """
    if x > y:
        return x
    return y

  = float(input("a = "))
  = float(input("b = "))
 rint("Max1:", max1(a, b))
 rint("Max2:", max2(a, b))
```

```
  = 17
  = 13
 ax1: 17.0
 ax2: 17.0
```

Can you find a and b such that these two functions return different values?

Docstrings

What is the use of these long explanations at the beginning of each function?

They give valuable information to anyone using your function, without them being required to read the code of your function. If this so-called *docstring* precisely defines the input and output of your function, and what it does, anyone can use that function as a *black-box*. This is extremely powerful! For example, let's see what `max1` from above does:

In [9]:

```
 rint(max1.__doc__)
```

```
    Takes comparable values `x` and `y` as arguments.

    Returns the larger of the two, implemented with if-else.
```

There is also a prettier version:

In [10]:

```
rom pydoc import help
elp(max1)
```

```
Help on function max1 in module __main__:

max1(x, y)
    Takes comparable values `x` and `y` as arguments.

    Returns the larger of the two, implemented with if-else.
```

Doctrings are also commonly used to automatically generate a referrence manual for your code, using - for example - Sphinx (http://sphinx-doc.org/).

Technically, docstrings are not *required* by Python and the functions will do just fine without them. However, they are a standard part of any Python code.

For this reason, make sure to write meaningful docstrings in the headers of ALL your programs and ALL your functions.

To grasp how important the docstrings are, take a look at the list of projects using Sphinx (http://www.sphinx-doc.org/en/master/examples.html).

Learn and do more

The "language" used in docstrings is called *Markdown* and it is widely used for simple formatting of documents (for example, these lecture notes are written using Markdown). The documentation is easy to find, as are the editors for various platforms. Here are two on-line editors which working with various browsers:

- StackEdit (https://stackedit.io/)
- Dillinger (https://dillinger.io/)

But what if I am the only one who will read this code?

It doesn't matter. The *future you* (even as little as 6 months away) will likely not remember what the *present you* wrote. Between the two, there will be a lot of other work done, most of it probably unrelated to Python or even programming in general.

Always document your code! Do not waste time on thinking if it is needed or not. It is always needed.

Default values of arguments

Many times it is desirable to call the same function with different arguments (not just different values). For example, the function `range` (which we have seen earlier) can be called in several different ways:

- `range(m, n, s)` -- numbers $m, m+s, \ldots, m+sk$, where $k \in \mathbb{N}$ such that $m + sk < n \leq m + s(k+1)$.
- `range(m, n)` -- equivalent of `range(m, n, 1)`. We say that the *default value* of `step` is 1.
- `range(n)` -- equivalent of `range(0, n)` or `range(0, n, 1)`. In other words, the default value of `start` is 0.

The most obvious reason to do this is convenience: why calling `range(0, n, 1)` every few lines of the code if we can just write `range(n)` each time?

But there are other reasons as well. Instead of just using some fixed value as the default one, a function can perform complex tasks to determine what the value should be.

For example, we could construct a function `data(name)` that retrieves some data (phone number, e-mail address, number of selfies uploaded to Facebook,...) about a person identified by `name`. If we omit `name` and just call `data()`, the function might check who is currently running the program and provide the data for that person. This is a *different* name for each person running the program, so not a simple matter of a constant.

Let us observe an example: a function that returns the sum of all digits that, when divided by `d`, give the reminder `r`.

In [1]:

```
def digit_sum(n, d, r):
    """
    Return the sum of all the digits of an integer `n` that,
    when divided by `d`, give the reminder `r`.

    Parameters
    ----------
    n : int
        A number whose sum is computed.
    d : int
        The number by which each digit is to be divided. If `d == 0`, an error oc
urs.
    r : int
        The desired reminder of the division. If `r < 0` or `r >= d`, the sum wil
 be zero.

    Returns
    -------
    digit_sum : int
        The resulting sum of digits.
    """

    # Initializations
    res = 0 # Result (do not use "sum", because there is a built-in function with
hat name)
    n = abs(n) # Lose the sign

    while n > 0:
        digit = n % 10
        if digit % d == r:
            res += digit
        n //= 10

    return res
```

Note: Notice the long docstring? This is how they should usually be written (description of what the function does, followed by the list of parameters and return values with types and descriptions). To avoid making these lecture notes unnecessarily long, we will usually not write docstrings in so much detail, but you are encouraged to do so in your own code.

It is easy to call the above function. For example, to calculate the sum of all odd digits of a number, we call it as follows:

In [12]:

```
 = int(input("n = "))
rint(digit_sum(n, 2, 1))
```

```
n = 12345
9
```

However, it might make sense to choose default values for some of the parameters. Which ones exactly, depends on the intended uses. In other words, you decide by yourself what makes the most sense.

So, let us ask ourselves what would shorter calls to the function mean?

- `digit_sum()`: no arguments, not even `n`; hard to imagine anything useful;
- `digit_sum(n)`: this could mean "calculate the sum of **all** digits of `n`"; this is equivalent to `d = 1, r = 0`;
- `digit_sum(n, d)`: only `r` is ommited; it makes sense to interpret this as "calculate the sum of **all** digits of `n` that are divisible by `d`"; this is equivalent to `r = 0`.

Luckily, `r` has the same desired default value in both cases, so it is clear that `r = 0` makes the most sense. Finally, our function (minus the docstring) might look like this:

In [13]:

```
def digit_sum(n, d=1, r=0):
    """
    ...
    """

    # Initializations
    res = 0 # Result (do not use "sum", because there is a built-in function with
hat name)
    n = abs(n) # Lose the sign

    while n > 0:
        digit = n % 10
        if digit % d == r:
            res += digit
        n //= 10

    return res
```

As you can see, only the function's declaration was changed, and in a very intuitive way:

```
    def digit_sum(n, d=1, r=0):
```

Here, `d=1` means "if the second argument's value was not given, set it to 1".

A note on the writing style: Both `d = 1` and `d=1` are, of course, correct. However, it is customary to not use spaces in function arguments assignments.

Now, the calls to `digit_sum` can be:

In [14]:

```
  = int(input("n = "))
 rint("The sum of all digits of", n, "is", digit_sum(n))
 rint("The sum of all even digits of", n, "is", digit_sum(n, 2))
 rint("The sum of all odd digits of", n, "is", digit_sum(n, 2, 1))
```

```
n = 12345
The sum of all digits of 12345 is 15
The sum of all even digits of 12345 is 6
The sum of all odd digits of 12345 is 9
```

Limitations

All the parameters can be given default values. However, one rule must be observed:

> **All the parameters without a default value must appear before those that have a default value.**

In other words, **this is invalid:**

```
    ef digit_sum(n = 17, d = 1, r):
```

The reason is simple: what would `digit_sum(5, 1)` mean? While it is obvious that we want `r = 1`, it is unclear whether the value `5` should be assigned to `n` or `d`. With more parameters, much worse confusions could happen.

That's why the parameters' values are assigned in order: the first given value is assigned to the first parameter, the second value to the second parameter, etc. Once we run out of values in the call of the function, the rest of the parameters are assigned their default values.

If we have provided too few values when calling the function (for example, if we give no values when calling `digit_sum`), our function call will fail. For example:

In [15]:

```
 rint(digit_sum())
```

```
---------------------------------------------------------------------
-------
 ypeError                                 Traceback (most recent cal
  last)
 ipython-input-15-099bd565ae7b> in <module>()
----> 1 print(digit_sum())

 ypeError: digit_sum() missing 1 required positional argument: 'n'
```

Named arguments

This is Python-specific, but we mention it as it sometimes comes in handy. Consider a function:

In [16]:

```
def pfv(a, b, c=3, d=4, e=5):
    """
    Takes 5 parameters and prints them.
    The last three have a default value 3, 4, and 5 respectively.

    "pfv" stands for "print five values". :-)
    """
    print(a, b, c, d, e)
```

As we have seen before, these calls are acceptable:

In [17]:

```
fv(17, 19, 23, 29, 31)
fv(17, 19, 23, 29)
fv(17, 19, 23)
fv(17, 19)
```

```
17 19 23 29 31
17 19 23 29 5
17 19 23 4 5
17 19 3 4 5
```

However, in Python we *can* keep default values for `c` and `d` while still giving a non-default value to `e`. Here is one such call:

In [18]:

```
fv(17, 19, e=31)
```

```
17 19 3 4 31
```

We can even insist that some parameters must to be named:

In [19]:

```
def pfv(a, b, c=3, *, d=4, e=5):
    """
    ...
    """
    print(a, b, c, d, e)
```

Now, parameters after `c` must be named. Otherwise, an error occures:

In [20]:

```
fv(17, 19, 23, 31)
```

```
---------------------------------------------------------------------
-------
 ypeError                                 Traceback (most recent cal
  last)
 ipython-input-20-c32f29a5cdcf> in <module>()
----> 1 pfv(17, 19, 23, 31)

 ypeError: pfv() takes from 2 to 3 positional arguments but 4 were g
 ven
```

A proper call looks like this:

In [21]:

```
fv(17, 19, 23, d=31)
```

```
17 19 23 31 5
```

This allows us to define different calls of the same function with the same number and types of parameters, while still treating them differently.

Function calls as conditions

What will the following code print?

In [22]:

```
def f(x):
    """
    Prints and returns `x`.
    """
    print(x)
    return x

f f(False) and f(True):
    print("The tooth is out there!")
lse:
    print("The cake is a lie.")
```

```
False
The cake is a lie.
```

Obviously, "`The cake is a lie.`" got printed because the `if` line evaluates to `if False and True` which is equivalent to `if False`, so the program executes the `else` part.

However, each call to `f` also prints the value that it is about to return. So, why is only `False` printed, and not `True` as well?

This happens because Python uses *lazy evaluation* (http://en.wikipedia.org/wiki/Lazy_evaluation) of Boolean expressions. This means that once the result is uniquely determined, no further evaluation is done.

On the shown example, we have `f(False) and f(True)`. This is what happens during the evaluation:

1. `f(False)` is executed. This results in printing of `False` and it returns `False`.
2. Now, Python has `False and f(True)`. However, `False and ANYTHING` is `False`, so Python doesn't bother with `f(True)` (whatever it returns, the whole expression would remain `False`).
   Therefore Python concludes that the whole expression is `False`, **without evaluating `f(True)`**, so `True` is never printed.
3. Python now runs `if False` and enters the `else` branch, printing "`The cake is a lie.`".

Similarly, `if f(True) or f(False)` would not run `f(False)` because `True or ANYTHING` is `True`, regardless of the value of `ANYTHING`.

Here is the whole set of `True`/`False` combinations for `and` and `or`:

In [23]:

```
def f(x):
    """
    Prints and returns `x`.
    """
    print("f:", x)
    return x

def g(x):
    """
    Prints and returns `x`.
    """
    print("g:", x)
    return x

f f(False) and g(False):
    print("Expression 1 is true.")
lse:
    print("Expression 1 is false")
f f(False) and g(True):
    print("Expression 2 is true.")
lse:
    print("Expression 2 is false")
f f(True) and g(False):
    print("Expression 3 is true.")
lse:
    print("Expression 3 is false")
f f(True) and g(True):
    print("Expression 4 is true.")
lse:
    print("Expression 4 is false")

f f(False) or g(False):
    print("Expression 5 is true.")
lse:
    print("Expression 5 is false")
f f(False) or g(True):
    print("Expression 6 is true.")
lse:
    print("Expression 6 is false")
f f(True) or g(False):
    print("Expression 7 is true.")
lse:
    print("Expression 7 is false")
f f(True) or g(True):
    print("Expression 8 is true.")
lse:
    print("Expression 8 is false")
```

```
f: False
Expression 1 is false
f: False
Expression 2 is false
f: True
g: False
Expression 3 is false
f: True
g: True
Expression 4 is true.
f: False
g: False
Expression 5 is false
f: False
g: True
Expression 6 is true.
f: True
Expression 7 is true.
f: True
Expression 8 is true.
```

Of course, these expressions can be much more complex than just singe function calls.

While it may look like a technical detail, lazy evaluation is often used without noticing it. For example:

In [24]:

```
  = int(input("a: "))
  = int(input("b: "))
f (b != 0 and a % b == 0):
   print(a, "is divisible by", b)
lse:
   print(a, "is NOT divisible by", b)
```

```
a: 17
b: 0
17 is NOT divisible by 0
```

Swapping the conditions in the above `if` can be catastrophic. Try the following code (only "`b != 0`" and "`a % b == 0`" are swapped; the rest is unchanged) for any `a` and for `b = 0`:

In [25]:

```
  = int(input("a: "))
  = int(input("b: "))
 f (a % b == 0 and b != 0):
    print(a, "is divisible by", b)
 lse:
    print(a, "is NOT divisible by", b)
```

```
a: 17
b: 0

---------------------------------------------------------------------
-------
 eroDivisionError                         Traceback (most recent cal
  last)
 ipython-input-25-af39d0be7924> in <module>()
      1 a = int(input("a: "))
      2 b = int(input("b: "))
----> 3 if (a % b == 0 and b != 0):
      4     print(a, "is divisible by", b)
      5 else:

 eroDivisionError: integer division or modulo by zero
```

Conclusion: When using complex expressions, put them in the order in which they can always be executed without raising any errors! Apart from the divisibility issues like the one above, this will also come in handy with lists and similar structures.

Of course, if the conditions can be run independently, always put those that compute faster first.

Scoping: how shy are the variables really?

It is important to know when and where our variables live, as well as when and where can we access them.

Consider the following example:

In [26]:

```
 = 1
 = 2
def f(x, y, z=17):
   print(" Inside #1:", x, y, z)
   y = 19
   print(" Inside #2:", x, y, z)

 = 3
rint("Outside #1:", x, y, z)
(x, y)
rint("Outside #2:", x, y, z)
(x, y, z)
rint("Outside #3:", x, y, z)
```

```
Outside #1: 1 2 3
 Inside #1: 1 2 17
 Inside #2: 1 19 17
Outside #2: 1 2 3
 Inside #1: 1 2 3
 Inside #2: 1 19 3
Outside #3: 1 2 3
```

Also:

In [27]:

```
 = 17
def f():
   print(x)
   x = 3
   print(x)

()
```

```
---------------------------------------------------------------------
-------
nboundLocalError                          Traceback (most recent cal
 last)
ipython-input-27-0bf68986ae5d> in <module>()
     5     print(x)
     6
----> 7 f()

ipython-input-27-0bf68986ae5d> in f()
     1 x = 17
     2 def f():
----> 3     print(x)
     4     x = 3
     5     print(x)

nboundLocalError: local variable 'x' referenced before assignment
```

But:

In [28]:

```
def f():
    print(x)

 = 17
()
```

```
17
```

Obviously, some variables "live" inside functions, some outside of them, some everywhere, ... This is called *scope*. Each programming language has its own rules of scoping. In Python, the rules are:

- *Global variables* are those that belong to no function, and they exist from their first assignment until the end of program's execution or until they are manually deleted.
- *Local variables* are those that belong to some function. They are created with the first assignment (including through function arguments) and they exist until the execution exits the function or until they are manually deleted.
  They lose their value between two calls to the same function!
- *Static variables* exist in some languages (C/C++, for example). They belong to a function and are not seen outside of it, but they do maintain their value between the function's calls. They do not exist in Python, but it is possible to simulate a very similar behaviour.

In Python, global variables are generally not accessible in functions. There are some exceptions to this rule:

- those that are never assigned a value (like `x` in the previous example), and
- those that are explicitly declared to be used from the global scope, using the `global` statement.

So, we can fix the above example with the assignment of global variable `x` in function `f`:

In [29]:

```
 = 17
def f():
    global x
    print(x)
    x = 3
    print(x)

rint("f:")
()
rint("x =", x)
```

```
f:
17
3
x = 3
```

The variables exist from their first assignment. So, this would work as well, even though the function is *written* before the assignment (it executes after it):

In [30]:

```
 ef f():
    global x
    print(x)
    x = 3
    print(x)

  = 17
 ()
```

```
17
3
```

When to use global variables in functions?

A short answer: never.

There are examples when using global variables is justifiable, but avoid them whenever you can. A function should be an independent entity that accepts input, does some work, and outputs the result.

Lambda functions

Lambda functions are a concept common to the functional programming languages, but it also exists in various forms in some other languages (MATLAB and JavaScript, for example).

The main idea is to easily define an *anonymous* function (the one without a name).

In Python, they are very simple: their only intent is to compute something and return it. For that reason, they do not contain more than a line of code and they don't contain `return` (it is implied).

For example:

In [31]:

```
  = lambda x, y: x + y
rint(s(2, 3))
```

```
5
```

Notice how `s` is invoked in the same manner as the "normal" functions we've seen before. This is because the "lambda" in the name refers only to their definition, but - in the end - they *are* "normal" functions. For example, the one above is equivalent to this:

In [32]:

```
 ef s(x, y):
    return x + y
 rint(s(2, 3))
```

```
5
```

Lamba functions will be more useful later on, when we cover lists. However, one can use them without lists, for example, to dynamically create a function:

In [33]:

```
 ef mkinc(inc):
    return lambda x: x + inc

 1 = mkinc(1)
 7 = mkinc(7)
 rint("17 + 1 =", i1(17))
 rint("17 + 7 =", i7(17))
```

```
17 + 1 = 18
17 + 7 = 24
```

This example also shows that functions in Python are just a (somewhat special type of) data. We have seen this before, when we didn't call a function, but instead we printed some of the data belonging to the function `max1` (its docstring):

```
    rint(max1.__doc__)
```

and when we gave it to another function as an argument:

```
    rom pydoc import help
    elp(max1)
```

Keep in mind the following difference:

- `f` is an object (that can be a function, as well as a "regular" variable);
- `f()` is an invocation of a function `f` (an error occures if `f` is something else).

This will be more important later on.

Boolean context

If a function is defined, its Boolean interpretation is `True`:

In [34]:

```
def f(x):
   print("X")

rint(bool(f))
```

```
True
```

For that reason, be careful to always call it properly, with parentheses:

In [35]:

```
def f():
   return False

f f:
   print("f evaluates to True")
lse:
   print("f evaluates to False")

f f():
   print("f() evaluates to True")
lse:
   print("f() evaluates to False")
```

```
f evaluates to True
f() evaluates to False
```

What more can be done?

- Functions can be defined inside other functions -- while it has its organizational uses, it is generally not that useful and will not be covered here;
- Recursions (https://en.wikipedia.org/wiki/Recursion_%28computer_science%29): functions that call themselves.

Code structure

Functions can be used to structure your code into independent blocks, which will make them much easier to write, read, and debug. You should therefore make good use of functions. It is also good practice to have as little code as possible outside functions, and we will learn more about this later when we discuss *modules.*

For the lab classes and tests, we will from now on require that all code (except module imports) is inside functions, including a `main()` function for testing the code. For example, if we are asked to write two functions `myfunction1()` and `myfunction2()` for computing the sum and product of two arguments, our program should have the following structure:

```
ef myfunction1(a, b):
  """
  Takes two input arguments a and b and returns their sum.
  """
  s = a + b
  return s

ef myfunction2(a, b):
  """
  Takes two input arguments a and b and returns their product.
  """
  s = a * b
  return s

ef main():
  a = 5
  b = 3
  print(myfunction1(a, b)) # hopefully outputs 8
  print(myfunction2(a, b)) # hopefully outputs 15

ain()
```

Note that we need to call the `main()` function in order for the testing commands to be executed.

A common mistake

It is a common mistake to include `input()` commands inside functions, even though the function's arguments already provide the required variables. **HENCE, DO NOT WRITE THIS:**

```
ef myfun(x):
  # THIS MAKES NO SENSE:
  x = input()
  # some computation with x
  # and possibly a return
```

Instead, if you really want the user to input a parameter for the function `myfun` to be called with, do this in the `main()` function like so:

```
ef myfun(x):
  # some computation with x
  # and possibly a return

ef main():
  x = input('give a parameter for myfun:')
  myfun(x)

ain()
```

If the value is only used for testing, you do not need the `input` call at all. Just test your functions with some fixed input parameters until you are sure that they are working for all cases. The advantage is that when you change your function `myfun()` later on, you can quickly execute all tests at once to see if they still run as expected. For example (the return values are made up, of course):

```
ef myfun(x):
  # some computation with x
  # and possibly a return

ef main():
  x = 17
  print('317:', myfun(x)) # myfun(17) should return 317
  x = 0
  print('28:', myfun(x)) # myfun(0) should return 28
  x = -5
  print('12:', myfun(x)) # myfun(-5) should return 12

ain()
```

The last example is a primitive form of unit testing (https://en.wikipedia.org/wiki/Unit_testing), where the developer (i.e., *you*) of a function also provides a collection of tests which can be executed quickly and conveniently. Unit tests are an essential part of any larger software project, and even in the basic form above it will help you write codes with less bugs more quickly.