Ch.3: Functions and branching

Hans Petter Langtangen[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Mar 4, 2015

We have used many Python functions

Mathematical functions:

```
from math import *
y = sin(x)*log(x)
```

Other functions:

```
n = len(somelist)
integers = range(5, n, 2)
```

Functions used with the dot syntax (called *methods*):

```
C = [5, 10, 40, 45]
i = C.index(10)        # result: i=1
C.append(50)
C.insert(2, 20)
```

What is a function? So far we have seen that we put some objects in and sometimes get an object (result) out of functions. Now it is time to write our own functions!

Functions are one of the most import tools in programming

- Function = a collection of statements we can execute wherever and whenever we want
- Function can take *input objects* (arguments) and produce output objects (returned results)
- Functions help to organize programs, make them more understandable, shorter, reusable, and easier to extend

Python function for implementing a mathematical function

The mathematical function

$$F(C) = \frac{9}{5}C + 32$$

can be implemented in Python as follows:

```
def F(C):
    return (9.0/5)*C + 32
```

Note:

- Functions start with `def`, then the name of the function, then a list of arguments (here `C`) - the *function header*
- Inside the function: statements - the *function body*
- Wherever we want, inside the function, we can "stop the function" and return as many values/variables we want

Functions must be called

A function does not do anything before it is called.

```
def F(C):
    return (9.0/5)*C + 32

a = 10
F1 = F(a)                                # call
temp = F(15.5)                           # call
print F(a+1)                             # call
sum_temp = F(10) + F(20)                 # two calls
Fdegrees = [F(C) for C in [0, 20, 40]] # multiple calls
```

(Visualize execution)

Note: `F(C)` produces (returns) a `float` object, which means that `F(C)` is replaced by this `float` object. We can therefore make the call `F(C)` everywhere a `float` can be used.

Functions can have as many arguments as you like

Make a Python function of the mathematical function

$$y(t) = v_0 t - \frac{1}{2}gt^2$$

```
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

# sample calls:
y = yfunc(0.1, 6)
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)
```

(Visualize execution)

Function arguments become local variables

```
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

v0 = 5
t = 0.6
y = yfunc(t, 3)
```

(Visualize execution)

Local vs global variables. When calling `yfunc(t, 3)`, all these statements are in fact executed:

```
t = 0.6  # arguments get values as in standard assignments
v0 = 3
g = 9.81
return v0*t - 0.5*g*t**2
```

Inside `yfunc`, `t`, `v0`, and `g` are *local variables*, not visible outside `yfunc` and desroyed after `return`.

Outside `yfunc` (in the main program), `t`, `v0`, and `y` are *global variables*, visible everywhere.

Functions may access global variables

The `yfunc(t,v0)` function took two arguments. Could implement $y(t)$ as a function of t only:

```
>>> def yfunc(t):
...     g = 9.81
...     return v0*t - 0.5*g*t**2
...
>>> t = 0.6
>>> yfunc(t)
...
NameError: global name 'v0' is not defined
```

Problem: `v0` must be defined in the calling program program before we call `yfunc`!

```
>> v0 = 5
>> yfunc(0.6)
1.2342
```

Note: `v0` and `t` (in the main program) are global variables, while the `t` in `yfunc` is a local variable.

Local variables hide global variables of the same name

Test this:

```
def yfunc(t):
    print '1. local t inside yfunc:', t
    g = 9.81
    t = 0.1
    print '2. local t inside yfunc:', t
    return v0*t - 0.5*g*t**2

t = 0.6
v0 = 2
print yfunc(t)
print '1. global t:', t
print yfunc(0.3)
print '2. global t:', t
```

(Visualize execution)

Question. What gets printed?

Global variables can be changed if declared global

```
def yfunc(t):
    g = 9.81
    global v0   # now v0 can be changed inside this function
    v0 = 9
    return v0*t - 0.5*g*t**2

v0 = 2   # global variable
print '1. v0:', v0
print yfunc(0.8)
print '2. v0:', v0
```

(Visualize execution)

What gets printed?

```
1. v0: 2
4.0608
2. v0: 9
```

What happens if we comment out `global v0`?

```
1. v0: 2
4.0608
2. v0: 2
```

`v0` in `yfunc` becomes a local variable (i.e., we have two `v0`)

Functions can return multiple values

Say we want to compute $y(t)$ and $y'(t) = v_0 - gt$:

```
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt

# call:
position, velocity = yfunc(0.6, 3)
```

Separate the objects to be returned by comma, assign to variables separated by comma. Actually, a tuple is returned:

```
>>> def f(x):
...     return x, x**2, x**4
...
>>> s = f(2)
>>> s
(2, 4, 16)
>>> type(s)
<type 'tuple'>
>>> x, x2, x4 = f(2)   # same syntax as x, y = (obj1, obj2)
```

Example: Compute a function defined as a sum

The function

$$L(x; n) = \sum_{i=1}^{n} \frac{1}{i} \left(\frac{x}{1+x} \right)^i$$

is an approximation to $\ln(1 + x)$ for a finite n and $x \geq 1$.

Corresponding Python function for $L(x; n)$:

```
def L(x, n):
    x = float(x)  # ensure float division below
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    return s

x = 5
from math import log as ln
print L(x, 10), L(x, 100), ln(1+x)
```

Returning errors as well from the L(x, n) function

We can return more: 1) the first neglected term in the sum and 2) the error ($\ln(1+x) - L(x;n)$):

```
def L2(x, n):
    x = float(x)
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
```

Functions do not need to return objects

```
def somefunc(obj):
    print obj

return_value = somefunc(3.4)
```

Here, `return_value` becomes `None` because if we do not explicitly return something, Python will insert `return None`.

Example on a function without return value

Make a table of $L(x;n)$ vs. $\ln(1+x)$:

```
def table(x):
    print '\nx=%g, ln(1+x)=%g' % (x, log(1+x))
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L2(x, n)
        print 'n=%-4d %-10g  (next term: %8.2e  '\
              'error: %8.2e)' % (n, value, next, error)
```

No need to return anything here - the purpose is to print.

```
x=10, ln(1+x)=2.3979
n=1    0.909091    (next term: 4.13e-01  error: 1.49e+00)
n=2    1.32231     (next term: 2.50e-01  error: 1.08e+00)
n=10   2.17907     (next term: 3.19e-02  error: 2.19e-01)
n=100  2.39789     (next term: 6.53e-07  error: 6.59e-06)
n=500  2.3979      (next term: 3.65e-24  error: 6.22e-15)
```

Keyword arguments are useful to simplify function calls and help document the arguments

Functions can have arguments of the form `name=value`, called *keyword arguments*:

```
def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
    print arg1, arg2, kwarg1, kwarg2
```

Examples on calling functions with keyword arguments

```
>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
>>     print arg1, arg2, kwarg1, kwarg2

>> somefunc('Hello', [1,2])   # drop kwarg1 and kwarg2
Hello [1, 2] True 0            # default values are used

>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0              # kwarg2 has default value

>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi           # kwarg1 has default value

>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi              # specify all args
```

If we use `name=value` for *all* arguments *in the call*, their sequence can in fact be arbitrary:

```
>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[2])
Hi [2] 6 Hello
```

How to implement a mathematical function of one variable, but with additional parameteres?

Consider a function of t, with parameters A, a, and ω:

$$f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)$$

Possible implementation. Python function with t as positional argument, and A, a, and ω as keyword arguments:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)

v1 = f(0.2)
v2 = f(0.2, omega=1)
v2 = f(0.2, 1, 3)  # same as f(0.2, A=1, a=3)
v3 = f(0.2, omega=1, A=2.5)
v4 = f(A=5, a=0.1, omega=1, t=1.3)
v5 = f(t=0.2, A=9)
v6 = f(t=0.2, 9)   # illegal: keyword arg before positional
```

Doc strings are used to document the usage of a function

Important Python convention: Document the purpose of a function, its arguments, and its return values in a *doc string* - a (triple-quoted) string written right after the function header.

```
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).

    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: a, b (floats) for the line (y=a*x+b).
    """
    a = (y1 - y0)/(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Convention for input and output data in functions

- A function can have three types of input and output data:
  - input data specified through positional/keyword arguments
  - input/output data given as positional/keyword arguments that will be modified and returned
  - output data created inside the function
- *All output data are returned, all input data are arguments*

```
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io7; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7
```

The function arguments are

- pure input: `i1`, `i2`, `i3`, `i6`
- input and output: `io4`, `io5`, `io7`

The main program is the set of statements outside functions

```
from math import *            # in main

def f(x):                     # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s

x = 2                         # in main
y = f(x)                      # in main
print 'f(%g)=%g' % (x, y)    # in main
```

The execution starts with the first statement in the main program and proceeds line by line, top to bottom.

`def` statements define a function, but the statements inside the function are not executed before the function is called.

Python functions as arguments to Python functions

- Programs doing calculus frequently need to have functions as arguments in other functions, e.g.,
  - numerical integration: $\int_a^b f(x)dx$
  - numerical differentiation: $f'(x)$
  - numerical root finding: $f(x) = 0$
- All three cases need f as a Python function `f(x)`

Example: numerical computation of $f''(x)$.

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

No difficulty with `f` being a function (more complicated in Matlab, C, C++, Fortran, Java, ...).

Application of the `diff2` function

Code:

```
def g(t):
    return t**(-6)

# make table of g''(t) for 13 h values:
for k in range(1,14):
    h = 10**(-k)
    print 'h=%.0e: %.5f' % (h, diff2(g, 1, h))
```

Output ($g''(1) = 42$):

```
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
h=1e-13: 66613381477.50939
```

Round-off errors caused nonsense values in the table

- For $h < 10^{-8}$ the results are totally wrong!
- We would expect better approximations as h gets smaller
- Problem 1: for small h we subtract numbers of approx equal size and this gives rise to round-off errors
- Problem 2: for small h the round-off errors are multiplied by a big number
- Remedy: use float variables with more digits
- Python has a (slow) float variable (`decimal.Decimal`) with arbitrary number of digits
- Using 25 digits gives accurate results for $h \leq 10^{-13}$
- Is this really a problem? Quite seldom - other uncertainies in input data to a mathematical computation makes it usual to have (e.g.) $10^{-2} \leq h \leq 10^{-6}$

Lambda functions for compact inline function definitions

```
def f(x):
    return x**2 - 1
```

The *lambda* construction can define this function in one line:

```
f = lambda x: x**2 - 1
```

In general,

```
somefunct = lambda a1, a2, ...: some_expression
```

is equivalent to

```
def somefunc(a1, a2, ...):
    return some_expression
```

Lambda functions can be used directly as arguments in function calls:

```
value = someotherfunc(lambda x, y, z: x+y+3*z, 4)
```

Example on using a lambda function to save typing

Old code:

```
def g(t):
    return t**(-6)

dgdt = diff2(g)
print dgdt
```

New, more compact code with lambda:

```
dgdt = diff2(lambda t: t**(-6))
print dgdt
```

If tests for branching the flow of statements

Sometimes we want to peform different actions depending on a condition. Example:

$$f(x) = \begin{cases} \sin x, & 0 \leq x \leq \pi \\ 0, & \text{otherwise} \end{cases}$$

A Python implementation of f needs to test on the value of x and branch into two computations:

```
from math import sin, pi

def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0

print f(0.5)
print f(5*pi)
```

(Visualize execution)

The general form if if tests

if-else (the else block can be skipped):

```
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

Multiple if-else.

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

Example on multiple branching

A piecewisely defined function.

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x < 1 \\ 2 - x, & 1 \le x < 2 \\ 0, & x \ge 2 \end{cases}$$

Python implementation with if-branching.

```
def N(x):
    if x < 0:
        return 0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0
```

Inline if tests for shorter code

A common construction is

```
if condition:
    variable = value1
else:
    variable = value2
```

This test can be placed on one line as an expression:

```
variable = (value1 if condition else value2)
```

Example:

```
def f(x):
    return (sin(x) if 0 <= x <= 2*pi else 0)
```

We shall write special *test functions* to verify functions

```python
def double(x):              # some function
    return 2*x

def test_double():          # associated test function
    x = 4
    exact = 8               # expected result from function double
    computed = double(x)
    success = computed == exact   # boolean value: test passed
    msg = 'computed %s, expected %s' % (computed, exact)
    assert success, msg
```

Rules for test functions:

- name begins with `test_`
- no arguments
- must have an `assert success` statement, where `success` is `True` if the test passed and `False` otherwise (`assert success, msg` prints `msg` on failure)

Test functions with many tests

```python
def double(x):              # some function
    return 2*x

def test_double2():         # test function
    tol = 1E-14             # tolerance for float comparison
    x_values = [3, 7, -2, 0, 4.5, 'hello']
    exact_values = [6, 14, -4, 0, 9, 'hellohello']
    for x, exact in zip(x_values, exact_values):
        computed = double(x)
        msg = '%s != %s' % (computed, exact)
        assert abs(exact - computed) < tol, msg
```

A test function will run silently if all tests pass. If one test above fails, `assert` will raise an `AssertionError`.

Why write test functions according to these rules?

- Easy to recognize where functions are verified
- Test frameworks, like `nose` and `pytest`, can automatically run *all* your test functions (in a folder tree) and report if any bugs have sneaked in

```
Terminal> nosetests -s .
Terminal> pytest -s .
```

Unit tests. A test function as `test_double()` is often referred to as a *unit test* since it tests a small unit (function) of a program. When all unit tests work, the whole program is supposed to work.

Summary of if tests and functions

If tests:

```
if x < 0:
    value = -1
elif x >= 0 and x <= 1:
    value = x
else:
    value = 1
```

User-defined functions:

```
def quadratic_polynomial(x, a, b, c)
    value = a*x*x + b*x + c
    derivative = 2*a*x + b
    return value, derivative

# function call:
x = 1
p, dp = quadratic_polynomial(x, 2, 0.5, 1)
p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)
```

Positional arguments must appear before keyword arguments:

```
def f(x, A=1, a=1, w=pi):
    return A*exp(-a*x)*sin(w*x)
```

A summarizing example for Chapter 3; problem

An integral

$$\int_a^b f(x)dx$$

can be approximated by *Simpson's rule*:

$$\int_a^b f(x)dx \approx \frac{b-a}{3n}\left(f(a) + f(b) + 4\sum_{i=1}^{n/2} f(a + (2i-1)h) \right.$$
$$\left. + 2\sum_{i=1}^{n/2-1} f(a + 2ih)\right)$$

Problem: make a function `Simpson(f, a, b, n=500)` for computing an integral of `f(x)` by Simpson's rule. Call `Simpson(...)` for $\frac{3}{2}\int_0^{\pi}\sin^3 xdx$ (exact value: 2) for $n = 2, 6, 12, 100, 500$.

The program: function for computing the formula

```
def Simpson(f, a, b, n=500):
    """
    Return the approximation of the integral of f
    from a to b using Simpson's rule with n intervals.
    """

    h = (b - a)/float(n)

    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
```

The program: function, now with test for possible errors

```
def Simpson(f, a, b, n=500):

    if a > b:
        print 'Error: a=%g > b=%g' % (a, b)
        return None

    # Check that n is even
    if n % 2 != 0:
        print 'Error: n=%d is not an even integer!' % n
        n = n+1  # make n even

    # as before...
    ...
    return integral
```

The program: application (and main program)

```
def h(x):
    return (3./2)*sin(x)**3

from math import sin, pi

def application():
    print 'Integral of 1.5*sin^3 from 0 to pi:'
    for n in 2, 6, 12, 100, 500:
        approx = Simpson(h, 0, pi, n)
        print 'n=%3d, approx=%18.15f, error=%9.2E' % \
              (n, approx, 2-approx)

application()
```

The program: verification (with test function)

Property of Simpson's rule: 2nd degree polynomials are integrated exactly!

```python
def test_Simpson():       # rule: no arguments
    """Check that quadratic functions are integrated exactly."""
    a = 1.5
    b = 2.0
    n = 8
    g = lambda x: 3*x**2 - 7*x + 2.5       # test integrand
    G = lambda x: x**3 - 3.5*x**2 + 2.5*x  # integral of g
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    success = abs(exact - approx) < 1E-14  # tolerance for floats
    msg = 'exact=%g, approx=%g' % (exact, approx)
    assert success, msg
```

Can either call `test_Simpson()` or run nose or pytest:

```
Terminal> nosetests -s Simpson.py
Terminal> pytest -s Simpson.py
...
Ran 1 test in 0.005s

OK
```