Ch.3: Functions and branching

Ole Christian Lingjærde, Dept of Informatics, UiO

September 4-8, 2017 (PART 2)

Today's agenda

- A small quiz
- Live-programming of exercises 3.20, 3.23, 3.28
- More about functions + branching

Quiz 1

If `a = ['A',['B',['B','C']]]` then which of the expressions below are equal to `B`?

- a[0]
- a[1][1]
- a[2][0]
- a[1][-2]
- a[-1][0]
- a[1][1][0]
- a[a.index('B')]
- a[len(a)-1][len(a)-1][0]

Quiz 2

Creating lists

- Create the list `a = ['A', 'A', ...., 'A']` of length 5000
- Create the list `b = ['A0', 'A1', ..., 'A4999']`

Equal or not?

Suppose `a = [0, 2, 4, 6, 8, 10]`. Which of the expressions below are equal to `True`?

- a[0] == a[-6]
- a[1] == a[-5]
- a[1:4] == [2, 4, 6, 8]
- a[1:4] == [a[i] for i in range(1,4)]
- a is a
- a[:] is a

Quiz 3

Suppose the following statements are performed:

```
a = [0, 1, 2, 3, 4]
b = a
b[0] = 50
print(a[0], b[0])
```

What is printed out here?

Quiz 4

Suppose the following statements are performed:

```
a = [0, 1, 2, 3, 4]
b = a[:]
b[0] = 50
print(a[0], b[0])
```

What is printed out here?

Quiz 5

Suppose we have defined a function

```
def h(x, y, z=0):
    import math
    res = x * math.sin(y) + z
    return res
```

Which of these function calls are allowed?

- r = h(0)
- r = h(0, 1)
- r = h(0, 1, 2)
- r = h(x=0, 1, 2)
- r = h(0, y=1)
- r = h(0, 1, z=3)
- r = h(0, 0, x=0)
- r = h(z=0, x=1)
- r = h(z=0, x=1, y=2)

Quiz 6

What is printed out here:

```
def myfunc(k):
    x = k * 2
    print('x = %g' % x)

x = 5
print('x = %g' % x)
myfunc(5)
print('x = %g' % x)
```

Exercise 3.20

Write functions

Three functions `hw1`, `hw2`, and `hw3` work as follows:

```
>>> print(hw1())
>>> Hello, World
>>>
>>> hw2()
>>> Hello, World
>>>
>>> print(hw3('Hello, ', 'World'))
>>> Hello, World
>>>
>>> print(hw3('Python ', 'function'))
>>> Python function
```

Write the three functions.

Filename: hw_func.

Exercise 3.23

Wrap a formula in a function

Implement the formula (1.9) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`.

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[0.76 \frac{T_0 - T_w}{T_y - T_w} \right].$$

The parameters ρ, K, c, and Tw can be set as local (constant) variables inside the function. Let t be returned from the function. Compute t for these conditions:

- Soft (Ty $<$ 70) and hard boiled (Ty $>$ 70)
- Small (M $=$ 47g) and large (M $=$ 67g) egg
- Fridge (T0 $=$ 4C) and hot room (T0 $=$ 25C).

Filename: egg_func.

Exercise 3.28

Find the max and min elements in a list

Given a list a, the `max` function in Python's standard library computes the largest element in a: max(a). Similarly, min(a) returns the smallest element in a.

Write your own max and min functions.

Hint: Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, set `max_elem` equal to that element. Use a similar technique to compute the minimum element.

Filename: maxmin_list.

More about functions: an example

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

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

Possible implementation in Python:

```
from math import pi, exp, sin

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

Observe that t is a positional argument, while A, a, and ω are keyword arguments. That gives us large freedom when calling the function:

```
v1 = f(0.2)                        # Only give t
v2 = f(0.2, omega=1)               # Change default value of omega
v3 = f(0.2, omega=1, A=2.5)        # Change default value of omega and A
v4 = f(A=5, a=0.1, omega=1, t=1.3) # Change all three parameters
v5 = f(0.2, 1, 2.5)                # Change default value of A and a
```

Even functions can be used as arguments in functions

In Python, functions are allowed to take functions as arguments. Thus we can "pass on" a function to another function.

Example: If we know how to compute $f(x)$ then we can use the following approximation to find numerically the 2nd derivative of $f(x)$ in a given point:

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

Python implementation:

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

Here, the first argument to `diff2(.)` is a function.

A small sidetrack

The function we just defined had one keyword argument `h=1E-6`. Is there any good reason to choose $h = 0.000001$ rather than a smaller or larger value?

- Mathematically, we expect the approximation to improve when h gets smaller.
- However, when we solve problems numerically we also need to take into account rounding errors.
- Some numerical problems are more sensitive to rounding errors than others, so in practice we may have to do a bit of trial and error.

The effect of changing the value of h

To study the effect of changing `h` we write a small program:

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

# Compute g''(t) for smaller and smaller values of h:
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
```

Rounding errors dominate for small h-values

For $h < 10^{-8}$ the results are totally wrong!

- **Problem 1:** for small h we subtract numbers of roughly equal size and this gives rise to rounding error.
- **Problem 2:** for small h the rounding error is divided by a very small number (h^2), which amplifies the error.

Possible solution: 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 $h \leq 10^{-13}$

However, higher accuracy is rarely needed in practice.

Functions vs. main program

The *main program* is the part of the program that is not inside any functions. In general:

- Execution starts with the first statement in the main program and proceeds line by line, top to bottom.
- Functions are only executed when they are called

Note: functions can be called from the main program *or* from a function. During program execution, this can sometimes result in long "chains" of function calls.

Recursive functions

- Functions can call other functions. A function can even call itself! In that case, the function is called *recursive*.
- For this to make sense, there must be some way of stopping the self-calls (or the program will never stop).

Example (allowed but makes little sense):

```
def f(x):
    print(x)
    f(x+1)
```

What is printed out from the call `f(0)`?

Recursive functions are an important topic in both mathematics and computer science. They can sometimes be used to solve problems very elegantly. This is the topic for more advanced courses.

Anonymous functions (lambda functions)

Sometimes a function just involves the calculation of an expression. In that case, we can use the *lambda construction* to define it.

Example: the function

```
def f(x,y):
    return x**2 - y**2
```

can be defined in just one line with the lambda construction:

```
f = lambda x, y: x**2 - y**2
```

Lambda functions can be used directly as arguments:

```
z = g(lambda x, y: x**2 - y**2, 4)
```

Can you guess why lambda functions are also called anonymous functions?

Documenting functions is important

To add a brief description (*doc string*) to a function, place it right after the function header and inside triple quotes.

Examples:

```
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 expression for a
    straight line y = a*x + b through two specified points.

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

If-tests

An if-test allows the program to take different actions depending on what the current state of the program is. An if-test thus branches (splits) the flow of actions.

Example: consider the function

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

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
```

General form of an if-test

Type 1 (if)

```
if condition:
    <block of statements, executed when condition==True>
```

Type 2 (if-else)

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

Type 3 (if-elif-else)

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

Example 1

A piecewise 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-elif-else:

```
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
```

Example 2

The following function counts how many times s occurs in a:

```
def count(s, a):
    cnt = 0
    for e in a:
        if e == s:
            cnt += 1
    return cnt
```

Example of use:

```
>>> count(5.3, [2.2, 6.6, 2.5, 5.3, 8.9, 5.3])
>>> 2
>>>
>>> count('Anna', ['Ola', 'Karianne', 'Anna', 'Jens'])
>>> 1
>>>
>>> count([1,2], [1, 5, [1,2], [1,2], 3])
>>> 2
```

Inline if-tests

Common construction:

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

More compact syntax with one-line if-else:

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

Example:

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

A very special form of if-test: assert

Sometimes in a program you want to stop program execution and give an error message if a condition is not true. For this purpose, we can can use the assert statement. General form:

```
assert condition, message
```

Example:

```
>>> x = 5
>>> assert x > 0, "x should be positive"  # Nothing happens
>>> x = -5
>>> assert x > 0, "x should be positive"  # Generates error message
Traceback (most recent call last):

  File "<ipython-input-30-c680011d20e2>", line 1, in <module>
    assert x > 0, "x should be positive"

AssertionError: x should be positive
```

Writing test functions

Suppose we have written a new function with some return values. To convince ourselves it works properly, we should try it for some input values and see if the result matches what we expect.

Note: the strategy above only works if we actually know what the answer *should* be. Often we know this for some input values.

Test strategy

- Write the new function.
- Write a *test function* that calls the new function with input values chosen so we know what the output should be.
- If the output from the new function differs from the expected output, we stop execution and print an error message.

Example

```
def sum3(a):                 # Find sum of every 3rd element in a
    res = sum([a[i] for i in range(0,len(a),3)])
    return res

def test_sum3():             # Associated test function
    """Call sum3(a) to check that it works."""
    a = [0,1,2,3,4,5]        # Some chosen input value
    expected = 3             # What the output should be
    computed = sum3(a)
    success = (computed == expected) # Did the test pass?
    message = 'computed %s, expected %s' % (computed, expected)
    assert success, message
```

Test functions with many tests

```
def sum3(a):                # Find sum of every 3rd element in a
    res = sum([a[i] for i in range(0,len(a),3)])
    return res

def test_sum3():            # Associated test function
    """Call sum3(a) to check that it works."""
    tol = 1E-14
    inputs = [[6], [6,1], [6,1,2], [6,1,2,3]]
    answers = [6, 6, 6, 9]
    for a, expected in zip(inputs, answers):
        computed = sum3(a)
        message = '%s != %s' % (computed, expected)
        assert abs(expected - computed) < tol, message
```

Recall that zip(a, b) creates pairs [a[i],b[i]]:

```
>>> zip(inputs, answers)
>>> [([6], 6), ([6, 1], 6), ([6, 1, 2], 6), ([6, 1, 2, 3], 9)]
```

More about test functions

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

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)

The optional `msg` parameter writes a message if the test fails.

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
- This is a very well established standard

```
Terminal> py.test -s .
Terminal> nosetests -s .
```

We recommend `py.test` - it has superior output.

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.

Comments on test functions

- Many find test functions to be a difficult topic
- The idea *is* simple: make problem where you know the answer, call the function, compare with the known answer
- Just write some test functions and it will be easy
- The fact that a successful test function runs silently is annoying - can (during development) be convenient to insert some print statements so you realize that the statements are run

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*:

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

where n is an even integer.
Problem: make a function `Simpson(f, a, b, n=500)` for computing an integral of `f(x)` by Simpson's rule.

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!

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

OK
```