Functions and branching

Foundation of programming (CK0030)

Francesco Corona

FdP

- ☻ Intro to variables, objects, modules, and text formatting
- ☻ Programming with WHILE- and FOR-loops, and lists
- ☹ **Functions and IF-ELSE tests**

- ☹ Data reading and writing
- ☹ Error handling
- ☹ Making modules

- ☹ Arrays and array computing
- ☹ Plotting curves and surfaces

FdP (cont)

Two fundamental and extremely useful programming concepts

- **Functions**, defined by the user
- **Branching**, of program flow

Functions

Functions and branching

Functions

The term **function** has a wider meaning than a mathematical function

Definition

- A `function` is a collection of statements that can be run wherever and whenever needed in the program

The `function` may accept input variables and may return new objects

- To influence what is computed by the statements in it

`Functions` help avoid duplicating bits of code (puts them together)

- A strategy that saves typing and makes it easier to modify code

`Functions` are also used to split a long program into smaller pieces

Functions (cont.)

Python comes with pre-defined `functions` (`math.sqrt`, `range`, `len`, ...)

- We discuss how to define own `functions`

Mathematical functions as Python functions

Functions

Mathematical functions as Python functions

We want to make a Python `function` that evaluates a math function

Example

Function $F(C)$ for converting degree Celsius C to Fahrenheit F

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

- The `function` (`F`) takes `C` (C) as its input argument

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

- It returns value `(9.0/5)*C + 32` ($F(C)$) as output

Mathematical functions as Python functions (cont.)

Definition

All Python `functions` begin with `def`, followed by the **function name**

- Inside parentheses, a comma-separated list of **function arguments**
- The argument acts as a standard variable inside the `function`

The statements to be performed inside the `function` must be indented

After the `function` it is common (not always) to **return** a value

- The **function output** value is sent out of the `function`

Mathematical functions as Python functions (cont.)

Example

Here `function name` is `F` (F), with only one `input argument` `C` (C)

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

The `return` value is evaluated `(9.0/5)*C + 32` (has no name)

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

The returned value is the evaluation of $F(C)$ (implicitly `F(C)`)

Remark

The `return` often (not necessarily) associates with the `function name`

Mathematical functions as Python functions (cont.)

Definition

The `def` line (`function name` and `arguments`) is the **function header**

- The indented statements are the **function body**

Example

```
def F(C):                                     # Function header
 return (9.0/5)*C + 32                        # Function (mini) block
```

Mathematical functions as Python functions (cont.)

Definition

To use a `function`, we must **call** or **invoke** it with `input arguments`

- The `function` will process the `input arguments`
- As a result, it will return an `output value`
- We need to store this value in a `variable`

Mathematical functions as Python functions (cont.)

Example

```
########################################
def F(C):                                       # T conversion function
 return (9.0/5)*C + 32                          # F(C)
########################################
```

The value returned from F(C) is an object, specifically a float object

- The call F(C) can be placed anywhere in a code where a float would be valid

```
temp1 = F(15.5)       # Store return value as variable (temp1)

a = 10                #    Given input argument 'a' (value 10)
temp2 = F(a)          # Store return value as variable (temp2)

                    # Given input argument 'a+1' (value 10 + 1)
print F(a+1)        # Print return value to screen (no storing)

sum_temp = F(10) + F(20)  # Two calls to get two output values
                          #    Combines output values and store
```

Mathematical functions as Python functions (cont.)

Example

Given a list Cdegrees of degrees Celsius, we want to compute a list of corresponding Fahrenheits using the F function in a list comprehension

```
########################################
def F(C):                               # T conversion function
 return (9.0/5)*C + 32                  # F(C)
########################################

Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35]

Fdegrees = [F(C) for C in Cdegrees]
```

Mathematical functions as Python functions (cont.)

Example

A slight variation of the `F(C)` function, named `F2(C)`, can be defined to return a formatted string instead of a real number

```
############################################################
def F2(C):                                                 #
 F_value = (9.0/5)*C + 32                                  #
 return '%.1f degrees Celsius correspond to '\             #
        '%.1f degrees Fahrenheit' % (C, F_value)           #
############################################################
```

```
>>> s1 = F2(21)
>>> print s1
    21.0 degrees Celsius correspond to 69.8 Fahrenheits
```

Remark

Note the `F_value` assignment inside the function

- We can create variables inside a function
- We can perform operations with them

Program flow

Functions

Program flow

Programmers must understand the sequence of statements in a program

- There are excellent tools that help build such understanding
- A **debugger** and/or the **Online Python Tutor**

A debugger should be used for all sorts of programs, large and small

- **Online Python Tutor** is an educational tool for small programs

Program flow (cont.)

Example

Program c2f.py contains a function F(C) and a while loop

- Print a table of converted degrees Fahrenheit

```
def F(C):
 F = 9./5*C + 32
 return F

dC = 10
C = -30
while C <= 50:
 print '%5.1f %5.1f' % (C, F(C))
 C += dC
```

A visual explanation of how the program is executed

- Go to **Online Python Tutor** (link/click me)

Forward button to advance, one statement at a time

- Observe the sequence of operations
- Observe the evolution of variables
- Observe, observe, observe, ...

Local and global variables

Functions

Local and global variables

Definition

Local variables are variables that are defined within a function

Remark

- Local variables are invisible outside functions

Example

```
>>> def F2(C):
 ... F_value = (9.0/5)*C + 32
 ... return '%.1f degrees Celsius correspond to '\
 ...        '%.1f degrees Fahrenheit' % (C, F_value)

>>> s1 = F2(21)
>>> s1
    '21.0 degrees Celsius correspond to 69.8 Fahrenheits'
```

In function F2(C), variable F_value is a local variable (inside a function), and a local variable does not exist outside the function

Local and global variables (cont.)

Example

An error message shows how the (main) program around (outside) function F2(C) is not aware of variable F_value

```
>>> def F2(C):
 ... F_value = (9.0/5)*C + 32
 ... return '%.1f degrees Celsius correspond to '\
 ...        '%.1f degrees Fahrenheit' % (C, F_value)

>>> c1 = 37.5
>>> s2 = F2(c1)

>>> F_value
 ...
 NameError: name 'F_value' is not defined
```

Local and global variables (cont.)

Remark

Local variables are created inside a function

- They are destroyed when leaving the function

Also input arguments are local variables

- They cannot be accessed outside the function

Example

The input argument to function F2, C, is a local variable

- We cannot access it outside the function

```
 ...
 ...

>>> C
 ...
NameError: name 'C' is not defined
```

Local and global variables (cont

Definition

`Variables` defined outside the `function` are **global variables**

`Global variables` are accessible everywhere in a program

- Also inside a `function`

Local and global variables (cont.)

```
#############################################################
def F2(C):                                                  #
 F_value = (9.0/5)*C + 32                                   #
 return '%.1f degrees Celsius correspond to '\              #
        '%.1f degrees Fahrenheit' % (C, F_value)            #
#############################################################
```

- `C` and `F_value` are `local variables`

```
>>> c1 = 37.5
>>> s2 = F2(c1)
```

- `c1` and `s2` (and `s1`) are `global variables`

Local and global variables (cont.)

```
>>> F_value
 ...
 NameError: name 'F_value' is not defined

>>> C
 ...
NameError: name 'C' is not defined
```

- Local variables cannot be accessed outside the function

Local and global variables (cont.)

Example

```
###############################################################
def F3(C):                                                    #
 F_value = (9.0/5)*C + 32                                     #
 print  'In F3: C=%s F_value=%s r=%s' % (C,F_value,r)         #
 return '%.1f Celsius correspond to '\                        #
        '%.1f Fahrenheit' % (C,F_value)                       #
###############################################################
```

Write out `F_value`, `C`, and a `global variable r` inside the `function`

```
>>> C = 60                            # Make a global variable, C
>>> r = 21                           # Another global variable, r

>>> s3 = F3(r)
    In F3: C=21 F_value=69.8 r=21

>>> s3
    '21.0 Celsius correspond to 69.8 Fahrenheit'

>>> C
    60
```

Local and global variables (cont.)

The example illustrates also that there are two different variables C

Example

One local variable, existing only when the program flow is inside F3

```
##############################################################
def F3(C):                                                   #
 F_value = (9.0/5)*C + 32                                    #
 print 'In F3: C=%s F_value=%s r=%s' % (C,F_value,r)         #
 return '%.1f Celsius correspond to '\                       #
        '%.1f Fahrenheit' % (C,F_value)                      #
##############################################################
```

One global variable, defined in the main (an int object), value 60

```
>>> C = 60
>>> r = 21
```

Local and global variables (cont.)

Example

```
##############################################################
def F3(C):                                                   #
 F_value = (9.0/5)*C + 32                                    #
 print  'In F3: C=%s F_value=%s r=%s' % (C,F_value,r)        #
 return '%.1f Celsius correspond to '\                       #
        '%.1f Fahrenheit' % (C,F_value)                      #
##############################################################
```

```
>>> C = 60
>>> r = 21
```

The value of the latter (`local`) `C` is given in the call to `function F3`

- When we refer to `C` in `F3`, we access the `local variable`
- Inside `F3`, `local variable C` shades `global variable C`

Local and global variables (cont.)

Remark

- `Local variables` hide `global variables`

Remark

Technically, `global variable` `C` can be accessed as `globals()['C']`

- This practice is deprecated, one should avoid working with `local` and `global variables` with the same names at the same time!

Local and global variables (cont.)

The general rule, when there are `variables` with the same name

1. Python first looks up the name among `local variables`
2. then among `global variables`
3. and, then among `built-in functions`

Local and global variables (cont.)

Example

```
print sum                      # sum is a built-in Python function
```

First line, no local variables are present, Python then searches for a global one named sum, cannot find any, checks in built-in functions

- It eventually finds a built-in function with name sum
- Printing sum returns <built-in function sum>

```
sum = 500                      # rebind name sum to an int object
print sum                               # sum is a global variable
```

Second line binds global name sum to an int object, when accessing sum in print statement, Python searches among global variables (still no local variables are present) and finds the one just defined

- The printout becomes 500

Local and global variables (cont.)

Example

```
print sum
sum = 500
print sum

def myfunc(n):
 sum = n + 1
 print sum                         # sum is a local variable
 return sum

sum = myfunc(2) + 1          # new value in global variable sum
print sum
```

Call `myfunc(2)` invokes a `function` where `sum` is a `local variable`

- `print sum` makes Python first search among `local variables`, and since `sum` is found there, the printout is now 3
- The printout is not 500, the value of `global variable sum`

Value of `local variable sum` is returned, added to 1, to form an `int object` (value 4), the object is then bound to `global variable sum`

Final `print sum` searches among `global variables`, finds one value 4

Local and global variables (cont.)

Remark

The values of `global variables` can be accessed inside `functions`

- Though their values cannot be changed
- Unless the variable is declared as global

Local and global variables (cont.)

Example

```
a = 20; b = -2.5                                  # global variables

def f1(x):
 a = 21                                  # this is a new local variable
 return a*x + b

print a                                                      # shows 20

def f2(x):
 global a                                        # a is declared global
 a = 21                                      # the global a is changed
 return a*x + b

f1(3); print a                                           # 20 is printed
f2(3); print a                                           # 21 is printed
```

Note that within `function f1`, `a = 21` creates a `local variable a`

- This does not change the `global variable a`

Multiple arguments

Functions

Multiple arguments

`Functions` `F(C)` and `F2(C)` are `functions` of one single variable `C`

- The `functions` take one `input argument` (`C`)

Yet, `functions` can have as many `input arguments` as needed

- Need to separate the `input arguments` by commas (`,`)

Multiple arguments (cont.)

Example

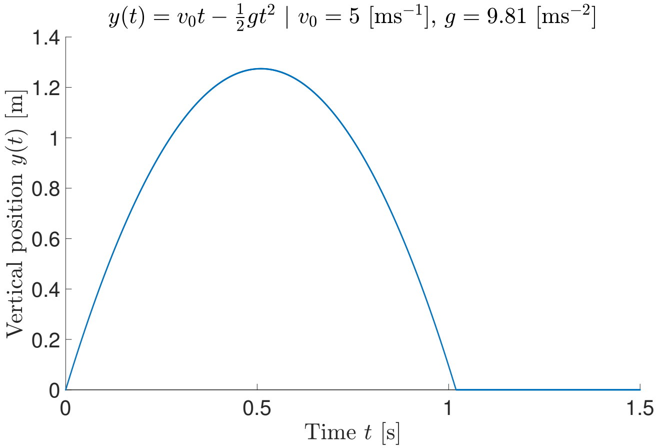
Consider the mathematical function

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

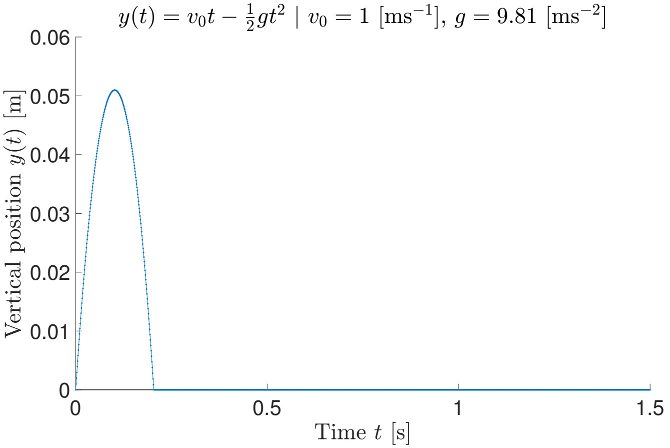
g is a fixed constant and v_0 is a physical parameter that can vary

- Mathematically, y is a function of one variable, t
- The function values also depend on the value v_0
- To evaluate y, we need values for both t and v_0

Multiple arguments (cont.)



Multiple arguments (cont.)



Multiple arguments (cont.)

A natural implementation would be a function with two arguments

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

Within the function, arguments t and v0 are local variables

Multiple arguments (cont.)

Example

We want to evaluate $y(t) = v_0 t - \frac{1}{2} g t^2$ for $v_0 = 6$ [ms^{-1}] at $t = 0.1$ [s]

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


y = yfunc(0.1  , 6)       # value1, value2
y = yfunc(0.1  , v0=6)    # value1, argument2=value2
y = yfunc(t=0.1, v0=6)    # argument1=value1, argument2=value2
y = yfunc(v0=6 , t=0.1)   # argument2=value2, argument1=value1
```

The possibility to write `argument=value` in the call facilitates reading and understanding the statement

Multiple arguments (cont.)

- With the `argument=value` syntax for all `arguments`, the sequence of the `arguments` is no longer important (we may put `v0` before `t`)
- When omitting the `argument=` part, the sequence of `arguments` in the call must match the sequence of `arguments` in the header

Remark

`argument=value` `arguments` must appear after all the `arguments` where only value is provided

- `yfunc(t=0.1, 6)` is illegal

Multiple arguments (cont.)

Whether `yfunc(0.1, 6)` or `yfunc(v0=6, t=0.1)` is used, `arguments` are automatically initialised as `local variables` within the `function`

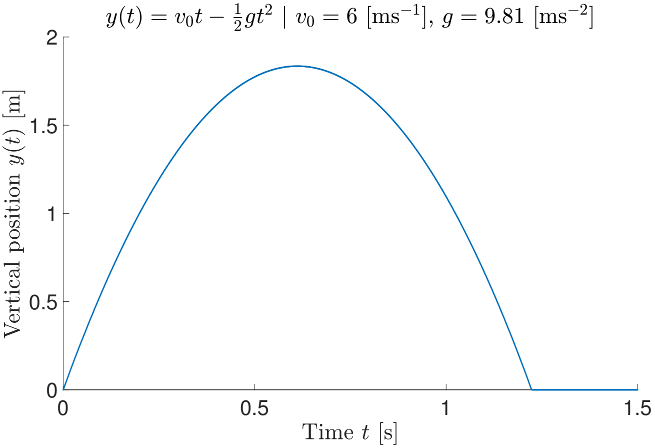
- **Initialisation** is the same as assigning values to `variables`

```
t = 0.1
v0 = 6

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

- Such `statements` are not visible in the code, yet a call to a `function` automatically initialises `arguments` this way

Multiple arguments (cont.)



Function argument v global variable

Functions

Function argument v global variable

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

Mathematically, y is a function of one variable, t, the implementation as Python function, yfunc, should also be a function of t only

Example

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

- v0 becomes a global variable, which needs be initialised outside function yfunc, before we attempt to call yfunc

Function argument v global variable (cont.)

Failing to initialise a `global variable` leads to an error message

Example

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

>>> yfunc(0.6)
    ...
    NameError: global name 'v0' is not defined
```

We need to define `v0` as a `global variable` prior to calling `yfunc`

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

Beyond math functions

Functions

Beyond math functions

So far, Python `functions` have typically computed some mathematical expression, but their usefulness goes beyond mathematical functions

- Any set of statements to be repeatedly executed under slightly different circumstances is a candidate for a Python `function`

Example

We want to make a list of numbers, starting from some value (`start`) and stopping at some other value (`stop`), with given increments (`inc`)

- Using variables `start=2`, `stop=8`, and `inc=2`, we would produce numbers 2, 4, 6, and 8

Beyond math functions (cont.)

Example

```python
def makelist(start, stop, inc):
 value = start
 result = []

 while value <= stop:
  result.append(value)
  value = value + inc

 return result

mylist = makelist(0, 100, 0.2)
print mylist       # will print 0, 0.2, 0.4, 0.6, ... 99.8, 100
```

- Function `makelist` has three arguments: `start`, `stop`, and `inc`
- Inside the `function`, the `arguments` become `local variables`
- Also `value` and `result` are `local variables`

In the surrounding program (`main`), we define one variable, `mylist`

- Variable `mylist` is a `global variable`

Beyond math functions (cont.)

Remark

`range(start, stop, inc)` does not make our `makelist` redundant

- `range` can only generate integers
- `makelist` can generate real numbers, too

Multiple returns

Functions

Multiple returns

Example

Suppose we are interested in some function $y(t)$ and its derivative $y'(t)$

$$y(t) = v_0 t - \frac{1}{2} g t^2$$
$$y'(t) = v_0 - g t$$

To get both $y(t)$ and $y'(t)$ from same `function` `yfunc`, we include both calculations and we separate variables in the `return` statement

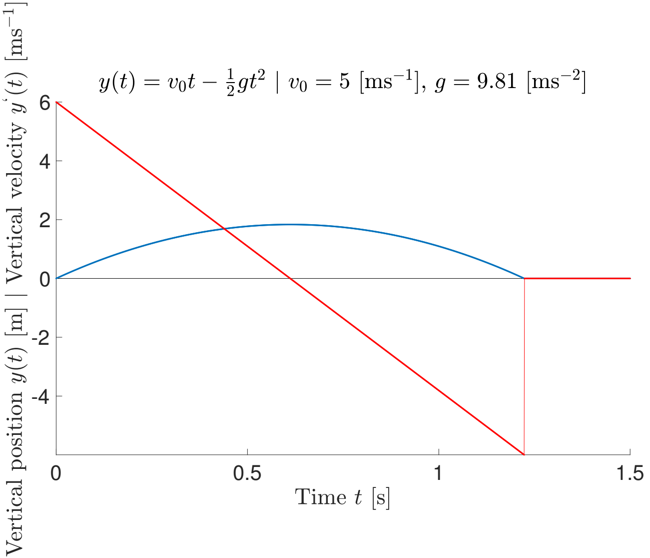
```
def yfunc(t, v0):
 g = 9.81
 y = v0*t - 0.5*g*t**2
 dydt = v0 - g*t
 return y, dydt
```

In the main, `yfunc` needs two names on LHS of the assignment operator

- Intuitively, as the function now returns two values

```
position, velocity = yfunc(0.6, 3)
```

Multiple returns (cont.)



Multiple returns (cont.)

Example

yfunc in the production of a formatted table of t , $y(t)$ and $y'(t)$ values

```
t_values = [0.05*i for i in range(10)]

for t in t_values:
 position, velocity = yfunc(t, v0=5)
 print 't=%-10g position=%-10g velocity=%-10g' % \
        (t, position, velocity)
```

Format %-10g prints a real number as compactly as possible (whether in decimal or scientific notation), within a field of width 10 characters

- The minus sign (-) after the percentage sign (%) leads to a number is left-adjusted in this field
- Important for creating nice-looking columns

Multiple returns (cont.)

```
t=0          position=0          velocity=5
t=0.05       position=0.237737   velocity=4.5095
t=0.1        position=0.45095    velocity=4.019
t=0.15       position=0.639638   velocity=3.5285
t=0.2        position=0.8038     velocity=3.038
t=0.25       position=0.943437   velocity=2.5475
t=0.3        position=1.05855    velocity=2.057
t=0.35       position=1.14914    velocity=1.5665
t=0.4        position=1.2152     velocity=1.076
t=0.45       position=1.25674    velocity=0.5855
```

Multiple returns (cont.)

Remark

Functions returning multiple, comma-separated, values returns a tuple

Example

```
>>> 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)                    # store in separate variables
```

Remark

Storing multiple returns in separate variables, as in the last line, is the same as storing list- (or tuple-) elements in separate variables

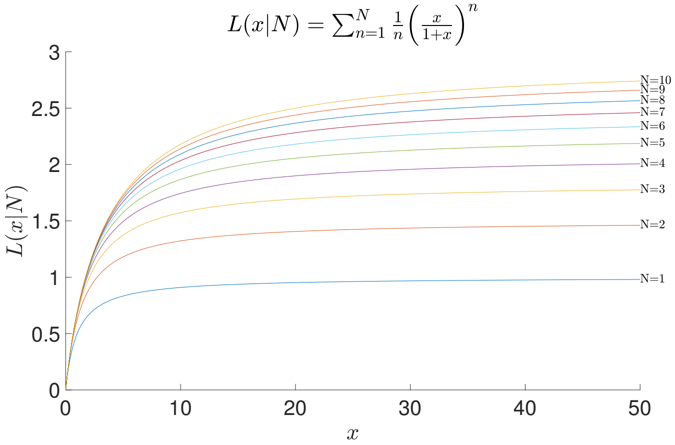
Summation
Functions

Summation

Example

Suppose we are interested in creating a `function` to calculate the sum

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



Summation

To compute the sum, a loop and add terms to an accumulation variable

- We performed a similar task with a `while loop`

Example

Summations with integer counters (like i) are normally (often) implemented by a `for-loop` over the `i` counter

$$\sum_{i=1}^{n} i^2$$

```
s = 0
for i in range(1, n+1):
 s += i**2
```

Summation (cont.)

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

```
s = 0
for i in range(1, n+1):
 s += (1.0/i)*(x/(1.0+x))**i
```

Observe the terms `1.0` used to avoid integer division

- `i` is an `int object` and `x` may also be an `int`

Summation (cont.)

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

We want to embed the computation of the sum in a Python `function`

- `x` and `n` are the `input arguments`

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

- The sum `s` is the `return output`

Summation (cont.)

It can be shown that $L(x; n)$ is an approximation to $\ln(1 + x)$ for a finite n and for $x \geq 1$, with the approximation becoming exact in the limit

$$\lim_{n \to \infty} L(x; n) = \ln(1 + x)$$

Instead of having `L` return only the value of the sum `s`, it would be also interesting to return additional information on the approximation error

Summation (cont.)

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

- The size of the terms decreases with `n`, the first neglected term (`n+1`) is bigger than all the remaining terms (for `n+2,n+3,...`), but not necessarily bigger than their sum
- The first neglected term is hence an indication of the size of the total error, we may use this term as a rough estimate of the error

Summation (cont.)

We return the exact error (we calculate the `log` function by `math.log`)

Example

```
###############################################################
def L2(x, n):                                                 #
 s = 0                                                        #
 for i in range(1, n+1):                                      #
  s += (1.0/i)*(x/(1.0+x))**i
                                #
 value_of_sum = s                                            #
 first_neglected_term = (1.0/(n+1))*(x/(1.0+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:
value, approximate_error, exact_error = L2(x, 100)
```

No returns

Functions

No returns

Sometimes a `function` can be defined to performs a set of statements

- Without necessarily computing objects returned to calling code

In such situations, the `return statement` is not needed

No returns (cont.)

The example shows the concept of `function` without `return values`

Example

- A table of the accuracy of the $L(x; n)$ approximation to $\ln(1 + x)$

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

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 returns (cont.)

```
>>> table(10)
 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)


>>> table(1000)
 x=1000, ln(1+x)=6.90875
 n=1     0.999001 (next term: 4.99e-01 error: 5.91e+00)
 n=2     1.498    (next term: 3.32e-01 error: 5.41e+00)
 n=10    2.919    (next term: 8.99e-02 error: 3.99e+00)
 n=100   5.08989  (next term: 8.95e-03 error: 1.82e+00)
 n=500   6.34928  (next term: 1.21e-03 error: 5.59e-01)
```

- Error is an order of magnitude larger than the first neglected term
- Convergence is slower for larger values of x than smaller x

No returns (cont.)

Remark

For `functions` w/o `return statement`, Python inserts an invisible one

- The invisible return is named `None`
- `None` is a special object in Python that represents something we might think of as the 'nothingness'

No returns (cont.)

Example

Normally, one would call `function table` w/o assigning `return value`

Imagine we assign the `return value` to a variable

- `result = table(500)`, the result will refer to a `None object`

No returns (cont.)

The None value is often used for variables that should exist in a program, but where it is natural to think of the value as conceptually undefined

Remark

The standard way to test if an object obj is set to None or not reads

```
if obj is None:
 ...

if obj is not None:
 ...
```

No returns (cont.)

- The `is` operator tests if two names refer to the same object
- The `==` tests checks if the contents of two objects are the same

```
>>> a = 1
>>> b = a
>>> a is b                      # a and b refer to the same object
    True

>>> c = 1.0             # a and c do not refer to the same object
>>> a is c
    False

>>> a == c                      # a and c are mathematically equal
    True
```

Keyword arguments

Functions

Keyword arguments

The input arguments of a function can be assigned a default value

- These arguments can be left out in the call

This is how a such a function may be defined

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

Keyword arguments (cont.)

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

First args (here, `arg1` and `arg2`) are `ordinary/positional arguments`

Last two args (`kwarg1` and `kwarg2`) are `keyword/named arguments`

Each `keyword argument` has a name and an associated a default value

Example

```
>>> somefunc('Hello', [1,2])
    Hello [1, 2] True 0

>>> somefunc('Hello', [1,2], kwarg1='Hi')
    Hello [1, 2] Hi 0

>>> somefunc('Hello', [1,2], kwarg2='Hi')
    Hello [1, 2] True Hi

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

Keyword arguments (cont.)

Remark

`Keyword arguments` must be listed AFTER `positional arguments`

When ALL `input arguments` are explicitly referred to (`name=value`), the sequence is not relevant: `positional` and `keyword` can be mixed up

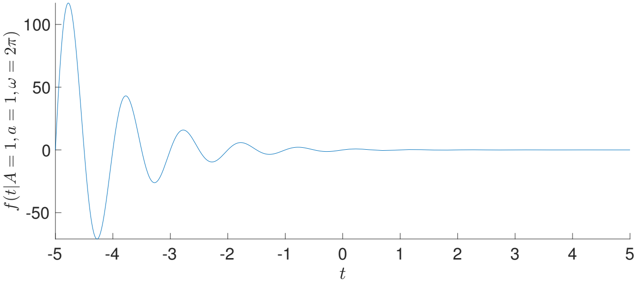
```
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[1,2])
    Hi [1, 2] 6 Hello
```

Keyword arguments (cont.)

Example

Consider some function of t also containing some parameters A, a and ω

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



Keyword arguments (cont.)

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

We code f as function of independent variable t, `ordinary argument`, with parameters A, a, and ω as `keyword arguments` with default values

```
from math import pi, exp, sin

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

Keyword arguments (cont.)

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

We can call function f with only argument t specified

```
v1 = f(0.2)
```

Other possible calls are listed below

```
v2 = f(0.2, omega=1)
v3 = f(1, A=5, omega=pi, a=pi**2)
v4 = f(A=5, a=2, t=0.01, omega=0.1)
v5 = f(0.2, 0.5, 1, 1)
```

Example

Consider $L(x;n)$ and functional implementations `L(x,n)` and `L2(x,n)`

$$L(x;n) = \sum_{i=1}^{n} \frac{1}{i}\Big(\frac{x}{1+x}\Big)^{i}, \text{ with } \lim_{n\to\infty} L(x;n) = \ln(1+x), \text{ for } x \geq 1$$

Instead of specifying the number n of terms in the sum,
we now specify a minimum tolerance ε in the accuracy

We can use the first neglected term as an estimate of the accuracy

- We add terms as long as the absolute value of the next term is greater than ε

Keyword arguments (cont.)

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

It is natural to provide a default value for ε

```
############################################################
def L3(x, epsilon=1.0E-6):                                 #
 x = float(x)                                              #
 i = 1                                                     #
 term = (1.0/i)*(x/(1+x))**i                               #
 s = term                                                  #
                                                           #
 while abs(term) > epsilon:                                #
  i += 1                                                   #
  term = (1.0/i)*(x/(1+x))**i                              #
  s += term                                                #

 return s, i                                              #
############################################################
```

Keyword arguments (cont.)

We make a table of the approximation error as ε decreases

```
#############################################################
def L3(x, epsilon=1.0E-6):                                  #
 x = float(x)                                               #
 i = 1                                                      #
 term = (1.0/i)*(x/(1+x))**i                                #
 s = term                                                   #
                                                            #
 while abs(term) > epsilon:                                 #
  i += 1                                                    #
  term = (1.0/i)*(x/(1+x))**i                               #
  s += term                                                 #

 return s, i                                               #
#############################################################

def table2(x):
 from math import log

 for k in range(4, 14, 2):
  epsilon = 10**(-k)
  approx, n = L3(x, epsilon=epsilon)
  exact = log(1+x)
  exact_error = exact - approx
  ...
```

Keyword arguments (cont.)

The output from calling `table2(10)` should look like

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

The `epsilon` estimate is about ten times smaller than the exact error

- regardless of the size of `epsilon`

Since `epsilon` follows the exact error over many orders of magnitude, we may view `epsilon` as a useful indication of the size of the error

Doc strings

Functions

Doc strings

There is a convention to augment `functions` with some documentation

- The **documentation string**, known as a **doc string**, should contain a short description of the purpose of the `function`
- It should explain what arguments and return values are
- Usually, right after the `def funcname:` line of definition

`Doc strings` are usually enclosed in triple double quotes `""""`

- This allows the string to span several lines

Doc strings (cont.)

Example

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

Example

```
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: coefficients a, b (floats) for the line (y=a*x+b).
 """

 a = (y1 - y0)/float(x1 - x0)
 b = y0 - a*x0
 return a, b
```

Doc strings (cont.)

To extract doc strings from source code use `funcname.__doc__`

Example

```python
print line.__doc__
```

```
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 (float objects).
x1, y1: another point on the line (float objects).
return: coefficients a, b (floats) for the line (y=a*x+b).
```

If function `line` is in a file `funcs.py`, we can run `pydoc funcs.line`

- Shows the documentation of function `line`
- Function signature and doc string

Doc strings (cont.)

Doc strings often contain interactive sessions, from the Python shell

- Used to illustrate how the function can be used

Example

```python
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 (float).
    x1, y1: another point on the line (float).
    return: coefficients a, b (floats) for the line (y=a*x+b).

    Example:
    >>> a, b = line(1, -1, 4, 3)
    >>> a
        1.3333333333333333
    >>> b
        -2.3333333333333333
    """

    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Functions (cont.)

It is a convention in Python that `function arguments` represent input data to the function, while `returned objects` represent output data

A general Python function looks like

Definition

```
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
 # modify io4, io5, io6; compute o1, o2, o3
 return o1, o2, o3, io4, io5, io7
```

- `i1`, `i2`, `i3` are `positional arguments`, input data
- `io4` and `io5` are `positional arguments`, input and output data
- `i6` and `io7` are `keyword arguments`, input and input/output data
- `o1`, `o2`, and `o3` are computed in the function, output data

Functions as arguments to functions

Functions

Functions as arguments to functions

We can have `functions` to be used as `arguments` to other `functions`

A math function $f(x)$ may be needed for specific Python `functions`

- Numerical root finding: Solve $f(x) = 0$, approximately
- Numerical differentiation: Compute $f'(x)$, approximately
- Numerical integration: Compute $\int_a^b f(x)\mathrm{d}x$, approximately
- Numerical solution of differential equations: $\dfrac{\mathrm{d}x}{\mathrm{d}t} = f(x)$

In such `functions`, function $f(x)$ can be used as `input argument` (`f`)

Functions as arguments to functions (cont.)

This is straightforward in Python and hardly needs any explanation

Remark

- In most other languages, special constructions must be used for transferring a function to another function as argument

Functions as arguments to functions (cont.)

Compute the 2nd-order derivative of some function $f(x)$, numerically

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}, \quad \text{with } h \text{ a small number}$$

Example

A Python function for the task can be implemented as follows

```
def diff2nd(f, x, h=1E-6):
 r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
 return r
```

f is, like other input arguments, a name, for a function object

Functions as arguments to functions (cont.)

```
##############################################################
def g(t):                                       # g(t) = t^(-6) #
 return t**(-6)                                                 #
##############################################################

##############################################################
def diff2nd(f, x, h=1E-6):                                      #
 r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)                      #
 return r                                                       #
##############################################################


t = 1.2
d2g = diff2nd(g, t)

print "g''(%f)=%f" % (t, d2g)
```

Functions as arguments to functions (cont.)

Asymptotically, the numerical approximation of the derivative becomes more accurate as $h \rightarrow 0$

Example

We show this property by making a table of the second-order derivatives

- $g(t) = t^{-6}$ at $t = 1$ as $h \rightarrow 0$

```
for k in range(1,15):
 h = 10**(-k)
 d2g = diff2nd(g, 1, h)
 print 'h=%.0e: %.5f' % (h, d2g)
```

Functions as arguments to functions (cont.)

The exact answer is $g''(t = 1) = 42$

```
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
h=1e-14: 0.00000
```

Computations start returning very inaccurate results for $h < 10^{-8}$

- For small h, on a computer, rounding errors in the formula blow up and destroy the accuracy
- Switching from standard floating-point numbers (`float`) to numbers with arbitrary high precision (`module decimal`) resolves the problem

The main program

Functions

The main program

In programs with functions, a part of the program is called **main**

- It is the collection of all statements outside the functions
- Plus, the definition of all functions

Example

```
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 main program here consists of the lines with comment in main

- The execution always starts with the first line in the main

The main program (cont.)

```python
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
```

When a function is encountered, its statements are used to define it

- Nothing is computed inside a function before it is called

Variables initialised in the main program become global variables

The main program (cont.)

```
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
```

1. Import `functions` from the `math module`
2. Define `function` `f(x)`
3. Define `x`
4. Call `f` and execute the function body
5. Define `y` as the value returned from `f`
6. Print a string

Lambda functions

Functions

Lambda functions

A one-line construction of functions used to make code compact

```
f = lambda x: x**2 + 4
```

This so-called **lambda function** is equivalent to the usual form

```
def f(x):
 return x**2 + 4
```

Definition

In general, we have

```
def g(arg1, arg2, arg3, ...):
 return expression
```

written in the form

```
g = lambda arg1, arg2, arg3, ...: expression
```

Lambda functions (cont.)

Lambda functions are used for function argument to functions

Example

Consider the diff2nd function used to differentiate $g(t) = t^{-6}$ twice

- We first make a g(t) then pass g as input argument to diff2nd

We skip the step of defining g(t) and use a lambda function instead

- A lambda function f as input argument into diff2nd

```
d2 = diff2nd(lambda t: t**(-6), 1, h=1E-4)
```

Lambda functions (cont.)

Remark

`Lambda functions` can also take `keyword arguments`

```
d2 = diff2nd(lambda t, A=1, a=0.5: -a*2*t*A*exp(-a*t**2), 1.2)
```

Branching

Functions and branching

Branching

The flow of computer programs often needs to branch

- if a condition is met, we do one thing;
- if it is not, we do some other thing

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

Implementing this requires a test on the value of x

Example

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

IF-ELSE blocks

Branching

IF-ELSE blocks

Definition

The general structure of the **IF-ELSE test**

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

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

- If `condition` is `True`, the program flow goes into the first block of statements, indented after the `if:` line
- If `condition` is `False`, program flow goes into the second block of statements, indented after the `else:` line

The blocks of statements are indented, and note the two-points

IF-ELSE blocks (cont.)

Example

```
if C < -273.15:
 print '%g degrees Celsius is non-physical!' % C
 print 'The Fahrenheit temperature will not be computed.'

else:
 F = 9.0/5*C + 32
 print F

print 'end of program'
```

- The two `print statements` in the **IF-block** are executed if and only if condition `C < -273.15` evaluates as `True`
- Otherwise, the execution skips the `print statements` and carries out with the computation of the statements in the **ELSE-block** and prints `F`

The `end of program` bit is printed regardless of the outcome

- This statement is not indented and it is neither part of the `IF-block` nor of the `ELSE-block`

IF-ELSE blocks (cont.)

The `else` part of the `IF-ELSE test` can be skipped

Definition

```
if condition:
 <block of statements>
<next statement>
```

Example

```
if C < -273.15:
 print '%s degrees Celsius is non-physical!' % C
F = 9.0/5*C + 32
```

The computation of `F` will always be carried out

- The statement is not indented
- It is not a part of the `IF-block`

IF-ELSE blocks (cont.)

Definition

With `elif` (for else if) several mutually exclusive `IF-test` are performed

```
if condition1:
 <block of statements>

elif condition2:
 <block of statements>

elif condition3:
 <block of statements>

else:
 <block of statements>
<next statement>
```

- This construct allows for multiple branching of the program flow

IF-ELSE blocks (cont.)

Example

Let us consider the so-called HAT function

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

Define a Python `function` that codes it

IF-ELSE blocks (cont.)

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

```
def N(x):
 if x < 0:
  return 0.0
 elif 0 <= x < 1:
  return x
 elif 1 <= x < 2:
  return 2 - x
 elif x >= 2:
  return 0.0
```

... or

IF-ELSE blocks (cont.)

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

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

Inline IF-tests

Branching

Inline IF-test

Variables are often assigned a value based on a boolean expression

This can be coded using a common `IF-ELSE test`

Definition

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

The equivalent one-line syntax, **inline IF-test**, for the snippet above

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

Inline IF-test (cont.)

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

Example

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

… or

Example

```python
f = lambda x: sin(x) if 0 <= x <= 2*pi else 0
```

Inline IF-test (cont.)

Remark

The `IF-ELSE test` cannot be used inside an `lambda function`, as it has more than one single expression

- `Lambda functions` cannot have statements
- A single expression only