Functions

Creative Commons Attribution Share-Alike 4.0 International

Functions

Splitting programs code into meaningful pieces (functions)
reduces time for debugging and maintaining

function: sequence of statements that perform a specific task
identified by a name

compound statement:

- **header**
  begins with keyword `def`
  ends with a colon (:)
- **body**
  indented statements
  includes a `return` statement

```
def function_name(formal parameters list):
    statements
    return returned expressions list
```

2 keywords: `def`, `return`
usual indentation: 4 spaces

Creative Commons Attribution Share-Alike 4.0 International

Functions

```
def function_name(formal parameters list):
    statements
    return returned expressions list
```

returned expressions list:
- computed results
- variables, values, expressions
- parenthesis not required
- comma separated items
- can be empty

formal parameters list:
- function is defined formally with them (data)
- variable names, no values, no expressions
- parenthesis required
- comma separated values
- can be empty

`return` statement: the last executed statement (not necessarily the bottom-most)
module: file with one or more functions

Creative Commons Attribution Share-Alike 4.0 International

Calling a function

a function is executed when it is **called (invoked):**
syntax of the **call statement** or **function call**:

```
variables list = function_name (actual parameters list)
```

variables list:
- are assigned the returned values
- variable names, no values, no expressions
- parenthesis not required

actual parameters list:
- specific values to apply the function
- variable names, values, or expressions
- parenthesis required

formal versus **actual** parameters lists
returned expressions versus variables lists } **positional** correspondence

Creative Commons Attribution Share-Alike 4.0 International

Exercise 1

Problem definition: A body moves with an uniformly accelerated rectilinear motion with an acceleration **a,** an initial velocity $\mathbf{v_0}$ and an initial position $\mathbf{x_0}$. Units are meters and seconds.

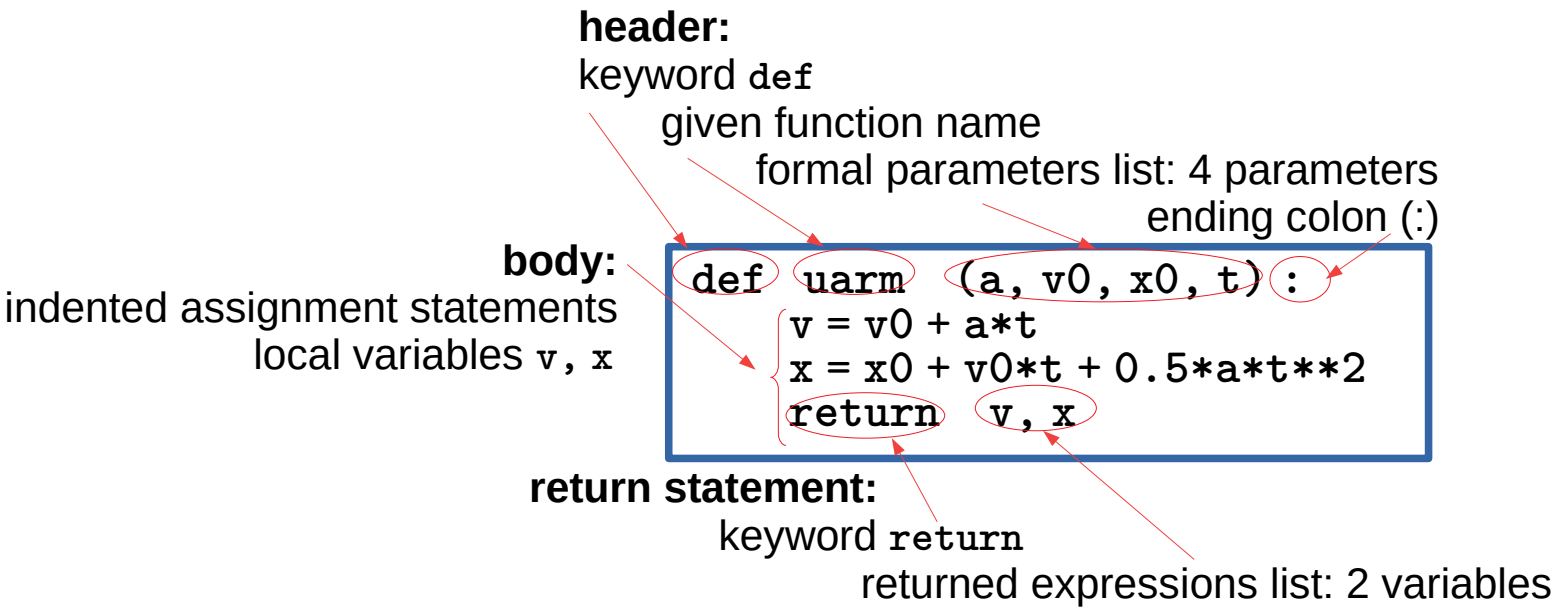
Write function `uarm` that takes these three parameters and a fourth one corresponding to a time, **t**, and returns the final position and velocity of the body **t** seconds after the motion start. Save this function in file `uarm.py`

Creative Commons Attribution Share-Alike 4.0 International

Exercise 1

Problem definition: A body moves with an uniformly accelerated rectilinear motion with an acceleration **a,** an initial velocity $\mathbf{v}_0$ and an initial position $\mathbf{x}_0$. Units are meters and seconds.

Write function `uarm` that takes these three parameters and a fourth one corresponding to a time, **t**, and returns the final position and velocity of the body **t** seconds after the motion start.



Creative Commons Attribution Share-Alike 4.0 International

Exercise 1

```
def uarm (a, v0, x0, t) :
    return  v0 + a*t,  x0 + v0*t + 0.5*a*t**2
```

Version with returned expressions:
body: only the return statement
returned expression list: 2 expressions separated by a comma

file `test-uarm.txt`

```
>>> from uarm import uarm
>>> uarm(1.1, 10.4, 3, 10) # actual parameters list: 4 values
(21.4, 162.0)
>>> uarm(2.1, 30/3.6, 100, 20) # actual parameters list: 3 values, 1 expression
(50.333333333333336, 686.6666666666667)
>>> vel, pos = uarm(2.1, 30/3.6, 100, 20) # 2 vars are assigned 2 returned expr
>>> round(vel,2), round(pos,2)            # rounded
(50.33, 686.67)
>>> a = 0.4
>>> b= 2.3
>>> c = 200
>>> d = 15
>>> vel, pos = uarm(a, b, c, d)  # actual parameters list: 4 variables
>>> round(vel,2), round(pos,2)
(8.3, 279.5)
```


Creative Commons Attribution Share-Alike 4.0 International

Flow of control

a function can call and be called by other functions

function A calls function B → function B gets the control
function B executes the return statement → function A recovers the control

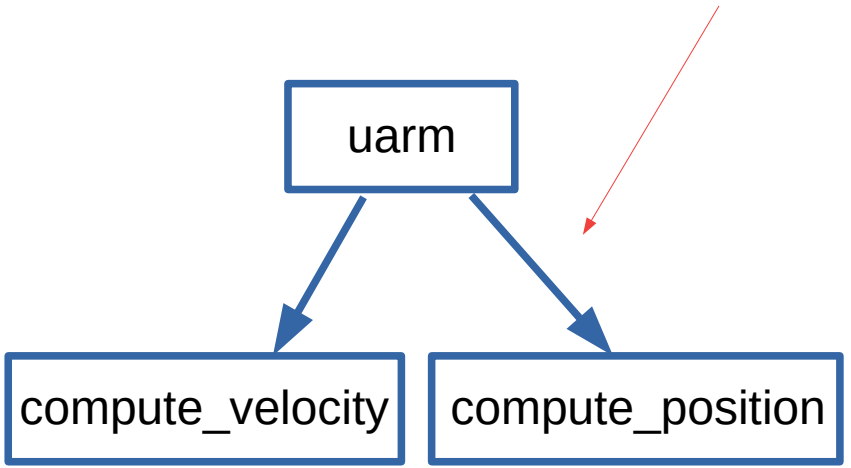
Version of exercise 1 using 2 auxiliary functions:

```
def uarm (a, v0, x0, t):
    v = compute_velocity(a, v0, t)
    x = compute_position(a, v0, x0, t)
    return v, x

def compute_velocity(a, v0, t):
    return v0 + a*t

def compute_position(a, v0, x0, t):
    return x0 + v0*t + 0.5*a*t**2
```

Flow of control represented by a directed (acyclic) graph



Creative Commons Attribution Share-Alike 4.0 International

Exercise 2

Problem definition: write function `is_a_root` that takes four parameters **a, b, c** and **x**, corresponding to the coefficients of a quadratic equation, $ax^2+bx+c = 0$, and a candidate root **x**, and returns `True` if **x** is a root of the quadratic equation and `False` otherwise.

```
>>> is_a_root(1, 2, 1, -1)
True
>>> is_a_root(1, 2, 1, 3)
False
```


Creative Commons Attribution Share-Alike 4.0 International

Exercise 2

Problem definition: write function `is_a_root` that takes four parameters **a, b, c** and **x**, corresponding to the coefficients of a quadratic equation, $ax^2+bx+c = 0$, and a candidate root **x**, and returns `True` if **x** is a root of the quadratic equation and `False` otherwise.

```
def is_a_root(a, b, c, x):
    return a*x**2+b*x+c == 0
```

Boolean expression:
`is_a_root` is a Boolean function

```
>>> is_a_root(1, 2, 1, -1)
True
>>> is_a_root(1, 2, 1, 3)
False
```


Creative Commons Attribution Share-Alike 4.0 International

Exercise 2

Consider the equation $x^2 - 10.88 = 0$

```
>>> import math
>>> round(math.sqrt(10.88), 2)
3.2984845004941286
>>> is_a_root(1, 0, -10.88, 3.29)
False
```

float comparisons:
precision problems
solved with a given precision `epsilon`

```
def is_a_root_2(a, b, c, x, epsilon):
    return abs(a*x**2+b*x+c) <= epsilon
```

```
>>> is_a_root_2(1, 0, -10.88, 3.3, 0.01) # using epsilon = 0.01
True
>>> is_a_root_2(1, 0, -10.88, 3.3, 0.001) # epsilon = 0.001 value too small
False
>>> is_a_root_2(1, 0, -10.88, 3.2984, 0.001)
True
```


Creative Commons Attribution Share-Alike 4.0 International