Computational Programming with Python
Unit 4 (cont.): More on Functions

Center for Mathematical Sciences, Lund University

Lecturer: Claus Führer, Alexandros Sopasakis
Unpacking Arguments

*Positional arguments* remind us of *lists*

*Keyword arguments* remind us of *dictionaries*

```python
data = [[1,2], [3,4]]
style = {'linewidth':3, 'marker':'o', 'color':'green'}
```

Star operators unpack these to form a valid parameter list

```python
plot(*data, **style)
```

* unpacks a list to positional arguments
  ** unpacks dictionaries to keyword arguments
Passing (tunneling) arguments

Also in the definition of functions you might find these constructs. This is often used to pass arguments through a function

```python
def outer(f, x, *args, **keywords):
    return f(x, *args, **keywords)

def inner(x, y, z, u):
    print(y, z)
    print(u)
    return x**2
```

A call

```python
L=[1, 2]
D={'u':15}
outer(inner, 3, *L, **D)
```

Equivalently:

```python
outer(inner, 3, 1, 2, u=15)
```

Note, the function outer cannot know how many arguments it needs to provide a full parameter list to the “inner” function f.
Return

The return statement returns a single object!

```python
def my_func(x):
    return 1, 2, 3, 4, 5, 6
```

What is the object that is returned here? Which type does it have? (see Unit 3)

Statements after the return statement are ignored:

```python
def my_func(x):
    if x > 0:
        return 1
    else:
        return -1
    z = 25  # ignored
```

```python
def my_func(x):
    return 1, 2, 3
    z = 25  # ignored
```
A function *without* a `return` statement returns `None`:

```python
def my_func(x):
    z = 2*x
    a = my_func(10.)

type(a)  # <type 'NoneType'>
a == None  # true
```
Functions are objects

Functions are objects, they can be deleted, reassigned, copied ...

def square(x):
    "Return the square of 'x'"   
    return x**2

square(4) # 16
sq = square # now sq is the same as square
sq(4) # 16
del square # 'square' doesn't exist anymore
print(newton(sq, .2)) # passing as argument
Partial Application

Partial application = closures

In mathematics we often “freeze” a parameter of a function:

\[ f_\omega(x) = f(x, \omega) = \sin(\omega x) \]

In Python there are many possibilities to do this, here is one ...

```python
def make_sine(omega):
    def f(x):
        return sin(omega*x)
    return f

fomega = make_sine(omega)
```

In order to understand this solution, recall

- the scope of a variable and references to variables out of scope
- functions are objects
Anonymous Functions: the `lambda` keyword

With $\lambda$-functions one has a handy tool for making one-line function definitions:

```python
f = lambda x: 3.*x**2 + 2.*x + 0.5
f(3)  # returns 33.5

g = lambda x,y: 3.*x - 2.*y
g(1,1) # returns 1.0
```

Example for a common application, compute $\int_0^1 x^2 + 5 \, dx$:

```python
import scipy.integrate as si
si.quad(lambda x: x**2+5,0,1)
```
Partial Applications and $\lambda$

$$f_\omega(x) = f(x, \omega) = \sin(\omega x)$$

.... now simply becomes

```python
omega = 3.
fomega = lambda x: sin(omega*x)
fomega(1.) # returns 0.14112...
```