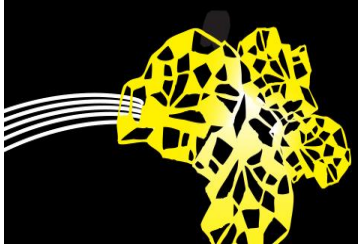
UNIVERSITY OF TWENTE.

Functions

Definition and use



1

This lecture's objectives

After this lecture, students can

- explain the concept of function
- illustrate the syntax of functions
- explain the use of variables in functions
- develop simple functions and functions with parameters
- structure programs using functions (after practicing …)

ITC

2

Today's contents

- what is a function?
- how do you define a function?
- what are arguments and parameters?
- how do you return a value from a function?

Functions

- A function is a portion of code that performs a specific task
- There are two types of function in Python:
  1. **Built-in** to the language
     - available directly
     - available within modules like the *Math* module
  2. **Custom** functions
     - customized routines created by the programmer

Function terminology

- Each function (may) have a name
  (Yes, there are occasionally functions that have no name, but this is something rather advanced.)
- A function (may) have parameters
- Each functions produces a result

Why do we have functions?

They are useful, and make programming life easier!

A function

1. groups statements
2. eliminates repeated code
3. cuts large programs into smaller, more understandable, parts
4. allows re-use of code

Built-in functions

- Python has a range of built-in functions that are part of the core language
- The full list is available in the Python Library Reference

https://docs.python.org/3/library/functions.html

Some examples:

- ***type()*** returns the data type of an arbitrary object

  print(type(2.0))

  <class 'float'>

- The ***len()*** function determines the length of a sequence

  print(len("hello"))

  5

Using (calling) a function

To use a function, you need to call it.

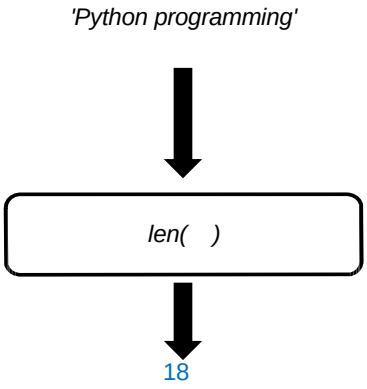
Example:

print(len('Python programming'))

18

- *'Python programming'* is the argument to *len()*
- *len('Python programming')* is the argument to *print()*

Input/output view



UNIVERSITY OF TWENTE.

9

Built-in functions: Conversion to another type

bool(n)
chr(n)
ord(c)
complex(real, imag)
dict(iterable)
float(n) hex(n)
int(n) list(iterable)
oct(x) repr(object)
round(n)
set(iterable)
str(object)
tuple(iterable)

UNIVERSITY OF TWENTE. 11

11

Modules

Modules are collections of code with a common purpose or area of application.

Each module contains a number of *pre-programmed functions (and classes)* which you are free to use in your programs.

Knowledge of which modules are available will help to code much more rapidly, consistently and productively.

Some useful modules

colorsys	conversions between color systems basic date and time
datetime	types and functions
decimal	decimal floating point arithmetic
ftp	FTP protocol client
math os	mathematical functions
pickle	miscellaneous operating system operations
random	object serialization (for storage)
re	pseudo-random number generator regular expressions
sys	system-specific parameters and functions

Mathematical functions

First, import the math module, informing the interpreter that you will use some of its components:

import math

A module is a file that contains a collection of related functions and classes (where a class is user-defined type)

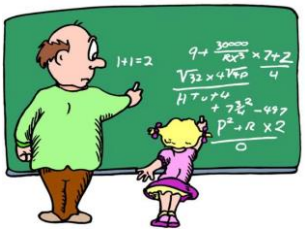
These functions must be used with the dot notation:

print(math.sin(math.radians(45)))

0.7071067811865475

help(math.radians)

Help on built-in function radians in module math: … (try it!)



The random module

```
import random
help(random)
...
help(random.random)
...
random.random()
0.5435454576
dir(random)

[……]
```

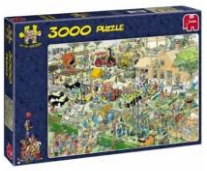
The *random()* function returns
random numbers between 0 and 1.

Its range does *not* include 1!

a list of all available attributes (constants and functions)

More modules

- Hundreds of modules are available in the Python installation https://docs.python.org/3.5/py-modindex.html



- Thousands of other modules can be installed; the Python Package index lists around 150,000 *packages* https://pypi.python.org/pypi

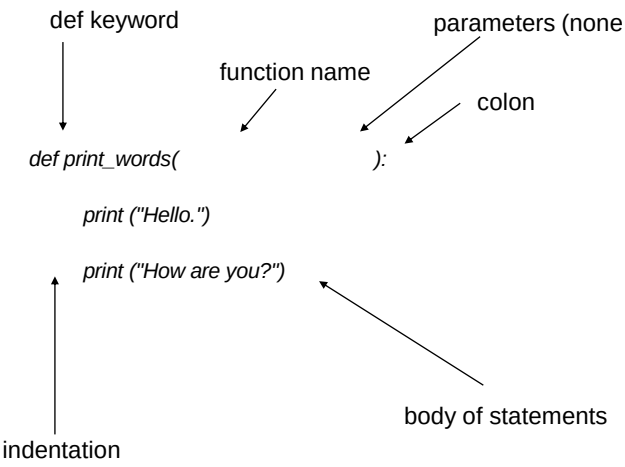
A package is a collection of modules organized as a folder

Function definition

The keyword **def** introduces a function definition. It must be followed by:

1. The function name
2. The list of parameters inside a pair of parentheses
3. A colon ':'
4. The statements that form the body of the function
5. These statements must be indented from def keyword!

Function definition example



Arguments

Parameters are formal slots in the *function definition*.
We use arguments to fill those slots when we *call (use) the function.*

```
def print_twice(x):
        print (x)
        print (x)
```

parameter

```
a = 'Hello World!'
print_twice(a)
```

argument

Hello World! Hello World!

Parameters are known only locally to the function

```
x = 23

print_twice(46)
46
46

print( x )
23
```

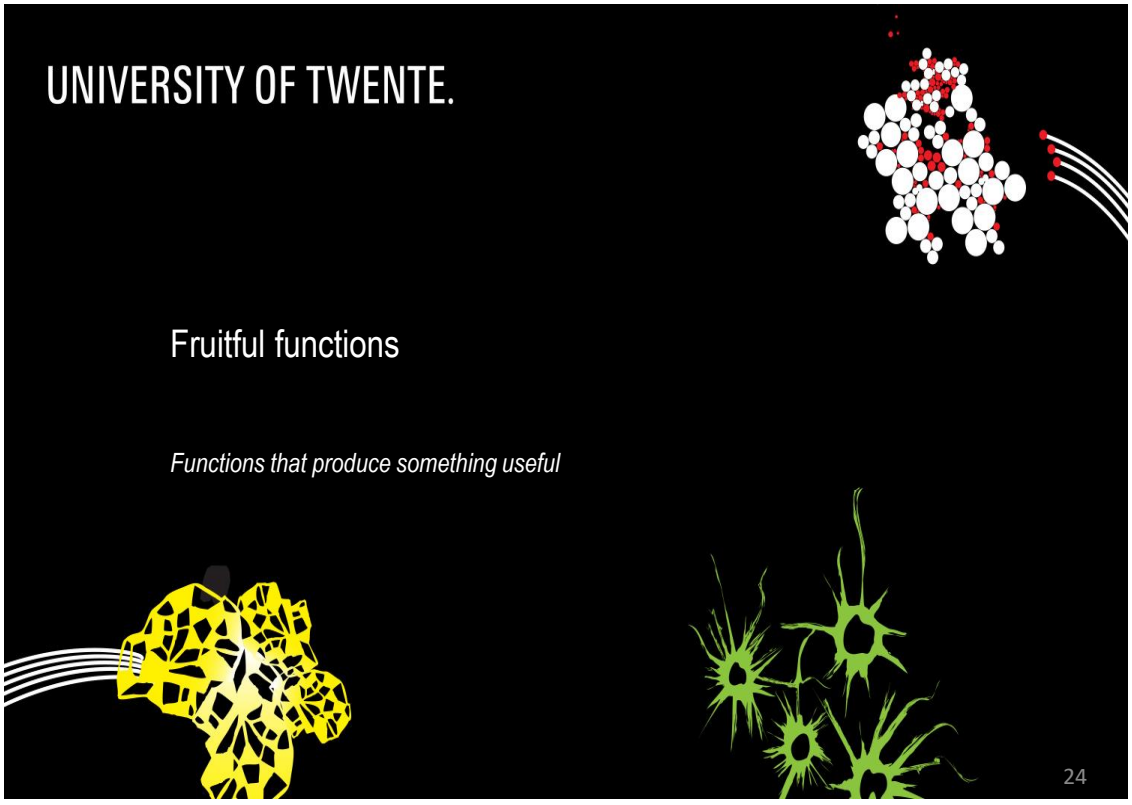
The function *print_twice* has used *x* as parameter. This will **not** clash; interpreter understands this is "*another x*."

That parameter *x* is said to be local to function *print_twice*.

23

UNIVERSITY OF TWENTE.

23



24

Return

A function may hand back a value by using the return keyword:

```
def double_it(x):
        return (x + x)
```

This is called a fruitful function

```
print ( double_it(23) )
46

a = double_it(23)
print( a )
46
```

Void functions

In Python, functions *always* return some value. That value is None when the function has no return statement.

```
def show_2(x):                  # this is called a void function
        print (x + x)

show_2 (23)
46
a = show_2(23)
print(a)
None
print(type(a))
<class 'NoneType'>
```

Recursion

Recursion is a technique to solve problems. It is based on the recognition that a problem can be solved by solving one or more smaller problems that are very similar.

What is the factorial of 12, i.e., *factorial(12)*? Of course, we can compute it as *12×11×10×⋯×1*.

But we can also compute it as *12×factorial(11)*. Here, *factorial(11)* is a "*smaller but similar problem*." Solve that one, and then you almost have solved the bigger problem also.

Recursion at work

For solving *factorial(11),* we will use *factorial(10),* and so forth. This is why it's called recursion.

Clearly, we should not continue towards
factorial(-109087330)
but stop at *factorial(1)* or perhaps *factorial(0).* Those two need no recursive approach, and we can simply define those cases to equal 1.

The latter two are the base cases where our solution "bottoms out."

Recursion in Python Functions

Can do this in Python. Define a function and call the same function in its function body.

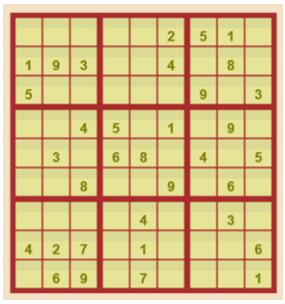
```
def factorial(n):
    if n==0 or n==1:
        return(1)
    else:
        return n * factorial(n-1)
factorial(5)
120
```

Recursion helps to write efficient programs using a minimal amount of code but *may cause infinite execution* if not written properly.
There may be risks. Two such risks are present in the above code. Which are they?

The above is a *classical recursion problem illustration*. Recursion does lend itself to non-trivial problems also.

Non-trivial recursion in Python

A less simple problem illustration is solving a Sudoku puzzle computationally.
Recursion can be the method that we use.



Sketch of recursive approach. The input is a partially filled puzzle like on the left. Python has a data structure that can hold such a partially filled puzzle (we will see later).

Suppose we call it P. It has empty slots, like the top-left cell.

Recursive approach to coding a sudoku solver would be

- find first empty cell (here: top-left), and identify that cell's possible fillers (here: 6, 7 and 8)
- if there are no possible fillers, P has no solution, so we give up on P
- otherwise, try each possible filler and recursively call the solver on the more filled P, which defines a "smaller problem," namely P with top-left cell filled with 6 (or 7, or 8)
- If no empty cell remains, we have found a/the solution

What you cannot do

Assign a value to a variable and use the variable *outside* of your function.
"Outside the function's scope."

Function variables are *local* to the function body.

```
def example_function(part1, part2):
    var1 = part1 + part2
    print (var1)

example_function(1,2)
3
print ( var1)
NameError: name 'var1' is not defined
```

… this is what the ***return*** statement is for

The pass statement

The pass statement does nothing.
It is useful when a statement is required by the syntax, but no code needs to be executed.

For example:

```
def lazy_thing(): pass

print (lazy_thing())
None
```

Debugging: Exit or quit function

The exit() or quit() function is useful in *debug mode*.
This function *stops* the execution of the code. No more code is interpreted.

```
def function(x):
        print('Hello world')
        exit('Stop here')
        print('Hello world 2')
```

You can use either exit() or quit(). They do the same!

Summary

- A function is code that helps perform repeated actions
- Python has a range of built-in functions and many external modules exist, covering a wide array of needs
- A function definition can have parameters; when called, each parameter slot is filled by an argument
- A function definition begins with the *de*f keyword, which is followed by the function body
- Every function returns a value (*None* if *return* keyword is not used)
- Function variables are local