Fenna Feenstra
Hanze *University of Applied Sciences*
Groningen

Bioinformatics data processing and –analysis using the scripting language Python

Introduction to programming with Python

Functions

Introduction

- When you want to repeat a set of calculations several times, you can copy-and-paste this code every time.
- However, if you decide to change the calculation, you will have to change it all over your script(s)
- This, and many other issues, can be prevented if you make use of **functions**

A first function

- a function is a block of code that has been given a name. Instead of copy the whole block of code again we just call the name of the code. Sometimes we have to pass data to a function and sometimes the function returns data.

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area
```

- In the code above we know it is a function because it has the keyword **def**. The function has the name **calc_area** and it needs two arguments: a value for the parameter **width** and a value for the parameter **length**. It returns the value of the variable **area**

Glossary - I

- Parameters are defined by the names that appear in a function definition. In the case below the function has two parameters, width and length:

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area
```

- Whereas arguments are the values actually passed to a function when calling it.

How to create a function

`def function_name():`
Keyword **def** defines a function with the given name

`(arg1, arg2, arg…)`
Parameter list that defines what types of arguments a function can accept

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area
```

This is the *body* of the function: the code that will be executed when you *call* it. Notice the *indentation*!

This function returns a value as outcome of the function which is called

How to call a function

- we can call a function by it's name and the arguments (values to be passed to the function parameters) , if it returns a value we can put that return value in a variable.

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
fieldsize = calc_area(10,20)
print(fieldsize)
```

How to call a function

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
fieldsize = calc_area(10,20)
print(fieldsize)
```

Two arguments are passed:10 and 20 . The 10 is passed to the parameter width and 20 is passed to parameter length.

How to call a function

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
fieldsize = calc_area(10,20)
print(fieldsize)
```

```
200
```

Since the parameters width and length are filled with the arguments 10 and 20 we can use them inside the function, like variables

The value 200 that was returned from the function is assigned to fieldsize

How to call a function

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
print(calc_area(10,20))
```

```
200
```

The value 200 that was returned from the function is now directly printed

How to call a function

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
a = 10
b = 20
print(calc_area(10,20))
print(calc_area(a,b))
print(calc_area(a, 20))
```

```
200
200
200
```

I can use either values or variables containing values

Parameters, arguments, variables and values (box versus content)

- Parameters are defined by the names that appear in a function definition. **(width, length)**
- Whereas arguments are the values actually passed to a function when calling it **(10,20)**
- Parameters can be used inside the function as variables

The result of a function I

- Not only can you pass a parameter value into a function, a function can also produce a value; the **return value**
- The **return** statement is followed by an expression which is evaluated. This is either a simple value, a variable containing a value or an expression resulting in a value.
- NB: All Python functions return the value **None** unless there is an explicit return statement with a value other than None.

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area
```

The result of a function II

```
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
a = 10
b = 20
fieldsize = calc_area(10,20)
print(fieldsize)
print(calc_area(10,20))
```

```
200
200
```

- If we call a function that returns a value we can catch this value in a variable by assigning the function call to a variable.
  fieldsize = calc_area(10,20)
- We also can use the result directly in another function as an argument, like the print function
- **print(calc_area(10,20))**

From body code to function and back

- A **return statement**, once executed, immediately terminates execution of a function, even if it is not the last statement in the function.
- A **return statement** causes execution to leave the current function and resume at the point in the code immediately after where the function was called.
- If the function does not have a return statement it leaves the function after the last statement of the function and resumes at the point in the code the function was called.

Returning a value

- Functions can only return a **single variable**. If you want to return more than one value, you've got to wrap them inside a
  - **List**
  - **Tuple**
  - **Dictionary**
  - Object (later presentations)
  - Generator function (later presentations)

Returning multiple values

```
# define the function
def abc_formula(a, b, c):
    D = b**2 - 4*a*c
    x1 = (-b + D**0.5) / (2*a)
    x2 = (-b - D**0.5) / (2*a)
    return [x1, x2]


# call the function
a = 1
b = 3
c = -4
outcome = abc_formula(a, b, c)
print(outcome)
print("solution x= ", outcome[0],  " or " ,outcome[1])
```

The function returns two values wrapped in a list. The list is one single **variable,** containing two values

```
[1.0, -4.0]
solution x=  1.0  or  -4.0
```

Outcome is a list as well

Returning multiple values

```
def basicStats( values ):
    total = 0
    min = float('inf')
    max = float('-inf')
    for num in values:
        total += num
        if num < min: min = num
        if num > max: max = num
    avg = float(total) / len(values)
    return {'min':min, 'max':max, 'avg':avg}

stats = basicStats([2.22, 3, 1.87, 5, 3, 4.56])
print('minimum=' + str(stats['min']))
```

```
minimum=1.87
```

Creating a float with 'inf' says: 'create the infinity value for this type'

This method takes a list of values and calculates the average, minimum and maximum values. It then returns a dictionary containing these values

To define or not to define

- Using functions is a good way to organize your code. With the keyword def we **define** a function. It is not called, just defined. The function code is **called** (executed) if we call it by it's functionname (without keyword def!).

```
#define the function calc_area
def calc_area(width, length):
    area = width*length
    return area

#call the function calc_area
print(calc_area(10,20))
print(calc_area(5,20))
```

```
200
100
```

Function elements

- Functions must
  - start with the keyword **`def`**
  - have a (legal) name
  - have an argument list, but it may be empty: **`( )`**
  - have a function body, but it may be simply the keyword **`pass`**

```
def my_function():
    pass
```

The minimal function; it does nothing at all

Function arguments

- Functions may receive arguments
- The arguments (values) are passed to the parameters which can be used as variables that live within the function as long as it runs

```
def reverse(seq):
    """Reverses a string of DNA"""
    rev = seq[::-1]
    return rev

rev_dna = reverse('ATTC')
print(seq)
```

'ATTC' is passed to seq but seq lives only within the function

```
---------------------------------------------------------------------
--
NameError                                 Traceback (most recent call las
t)
<ipython-input-28-728bf949b862> in <module>()
      5
      6 rev_dna = reverse('ATTC')
----> 7 print(seq)

NameError: name 'seq' is not defined
```

Function arguments and variables

- Functions may receive arguments
- Arguments are variables that live within the function as long as it runs

```
def reverse(seq):
    """Reverses a string of DNA"""
    rev = seq[::-1]
    return rev

rev_dna = reverse('ATTC')
print(rev_dna)
print(rev)
```

Also the variable 'rev' lives only within the function; the value 'CTTA' is returned, not the variable name rev

```
CTTA

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-2857490bc64c> in <module>()
      6 rev_dna = reverse('ATTC')
      7 print(rev_dna)
----> 8 print(rev)

NameError: name 'rev' is not defined
```

Function arguments

- If a function expects and argument and you do not provide one, you get an error

```
def reverse(seq):
    """Reverses a string of DNA"""
    rev = seq[::-1]
    return rev

rev_dna = reverse()
print(rev_dna)
```

The function is called without argument

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-24ca4c89ab3f> in <module>()
      4     return rev
      5
----> 6 rev_dna = reverse()
      7 print(rev_dna)

TypeError: reverse() missing 1 required positional argument: 'seq'
```

Function argument defaults

- If a function expects and argument and you do not provide one, you get an error - unless you provide a **default value**

```
def reverse(seq = 'ATCG'):
    """Reverses a string of DNA"""
    rev = seq[::-1]
    return rev

rev_dna = reverse()
print(rev_dna)
```

```
GCTA
```

This function defines a default value

Multiple function arguments

- A function can define as many arguments as you like, of whatever type you like

```
def printPersonalInfo( name, age, bmi ):
    if 2*bmi > age:
        print('Hello ' + name
        + ', you may want to work out a little')
    else:
        print('Hello ' + name
        + ', quite fit considering your age')

printPersonalInfo('Fenna', 46, 21.7)
```

```
Hello Fenna, quite fit considering your age
```

Multiple arguments and defaults

- If you want to provide *some* default values but not all, the arguments with default values should be the last one(s)

```
def printPersonalInfo( name, age, bmi = 30 ):
    if 2*bmi > age:
        print('Hello ' + name
        + ', you may want to work out a little')
    else:
        print('Hello ' + name
        + ', quite fit considering your age')

printPersonalInfo('Fenna', 46)
printPersonalInfo('Fenna', 46, 22)
```

```
Hello Fenna, you may want to work out a little
Hello Fenna, quite fit considering your age
```

Named parameter passing

- You can also specifically name your parameters when you call a function

```
def printPersonalInfo( age, bmi, name='John Doe' ):
    if 2*bmi > age:
        print('Hello ' + name
        + ', you may want to work out a little')
    else:
        print('Hello ' + name
        + ', quite fit considering your age')

printPersonalInfo( bmi=33.5, age=55, name='TooHeavy')
printPersonalInfo( bmi=33.5, age=55)
printPersonalInfo( 55, bmi=33.5, name='TooHeavy')
```

```
Hello TooHeavy, you may want to work out a little
Hello John Doe, you may want to work out a little
Hello TooHeavy, you may want to work out a little
```

If you do NOT name your parameters, they have to come first, and in the order declared in the function parameter list

Returning multiple values

```
def basicStats( values ):
    total = 0
    min = float('inf')
    max = float('-inf')
    for num in values:
        total += num
        if num < min: min = num
        if num > max: max = num
    avg = float(total) / len(values)
    return {'min':min, 'max':max, 'avg':avg}

stats = basicStats([2.22, 3, 1.87, 5, 3, 4.56])
print('minimum=' + str(stats['min']))
```

```
minimum=1.87
```

Creating a float with 'inf' says: 'create the infinity value for this type'

This method takes a list of values and calculates the average, minimum and maximum values. It then returns a dictionary containing these values

Built in

- python has some built in functions. You already used some. For instance:
  - print - prints the content of a string.
  - ord - translate a character to an ascii number
- you can call functions within functions (remember, it is just a piece of code you point to)

str functions

- there are several built in functions you can use for variables, like the data type string uses functions like
  - **.upper()** changes all the character of a string to capitals
  - **.strip()** remove leading and trailing spaces
  - **.join**(iterable) joins all the elements of an iterable in a string
- with **dir(str)** you learn more about all the functions

Scoping rules

- Before you started writing functions, all code was written at the top-level of a python script(module), so the names either lived in the module itself, or were built-ins that Python predefines (e.g., open)
- Functions provide a nested namespace (sometimes called a *scope*), which localizes the names they use, **such that names inside the function won't clash with those outside** (in a module or other function). We usually say that **functions def in a *local*** scope, and **modules define a *global*** scope.

Global and local

- Each module is a global scope—a namespace where variables created (assigned) at the top level of a module file live
- Every time you call a function, you create a new local scope—a namespace where variables names created inside the function usually live, but they do not exist outside the local space

Global versus Local

```
def reverse(seq):
    """Reverses a string of DNA"""
    rev = seq[::-1]
    return rev

rev_dna = reverse('ATTC')
print(rev_dna)
print(rev)
```

rev is defined in a function. So it is a local variable only exist in the local namespace

```
CTTA

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-2857490bc64c> in <module>()
      6 rev_dna = reverse('ATTC')
      7 print(rev_dna)
----> 8 print(rev)

NameError: name 'rev' is not defined
```

Scoping rules

- When you use an unqualified name inside a function, Python searches three scopes—the local (L), then the global (G), and then the built-in (B)—and stops at the first place the name is found.
- When you assign a name in a function (instead of just referring to it in an expression), Python always creates or changes the name in the local scope, unless it's declared to be global in that function.
- When outside a function (i.e., at the top-level of a module or at the interactive prompt), the scope is global

Best practice

- Code in functions to make your program more readable, maintainable and flexible
- Use functions with parameters, local variables and return values