Preparatory Course Informatics for Life Scientists

An Introduction to Python 3:
Control Flow and Scope

Philipp Thiel

September 13, 2022

Control Flow: Motivation

- As yet, we execute statements sequentially: one line after the other
- The **control flow** is linear: one direction, no alternative routes
- Programming gains power by:

  $\rightarrow$ **Manipulating the direction of the control flow**

  $\rightarrow$ **Choosing between alternative routes**

- Special statements allow us to influence the control flow
  - if statement: keywords **if**, **elif**, **else**
  - for loop: keyword **for**
  - while loop: keyword **while**
  - function definition: keyword **def**

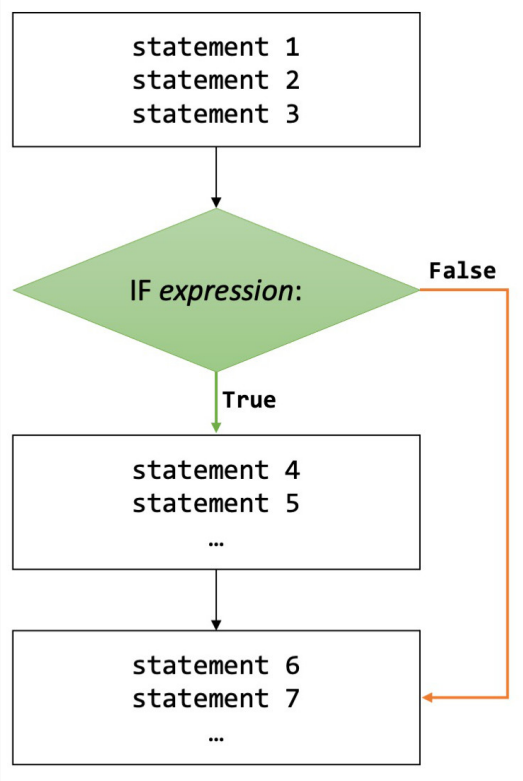
Control Flow: if Statement

- An **expression** is evaluated to either True or False (bool)

  → **True**: the statements inside the if-body are executed

  → **False**: the if-body is skipped and execution proceeds thereafter

- It is thus called a **conditional statement**



Control Flow: if Statement

myscript.py

```
1  seq = "ACTGGAGTCAGG"
2
3  if len(seq) == 0:
4      print("The DNA sequence is empty")
5
6  print("\n+++ sequence analysed +++")
```

```
1  $ python myscript.py
```

Control Flow: if Statement

myscript.py

```
1  seq = "ACTGGAGTCAGG"
2
3  if len(seq) == 0:
4      print("The DNA sequence is empty")
5
6  print("\n+++ sequence analysed +++")
```

```
1  $ python myscript.py
2
3  +++ sequence analysed +++
4  $
```

Lesson: if-body is skipped if expression evaluates to False

Control Flow: if Statement

myscript.py

```
1  seq = ""
2
3  if len(seq) == 0:
4      print("The DNA sequence is empty")
5
6  print("+++ sequence analysed +++")
```

```
1  $ python myscript.py
```

Control Flow: if Statement

myscript.py

```
1  seq = ""
2
3  if len(seq) == 0:
4      print("The DNA sequence is empty")
5
6  print("+++ sequence analysed +++")
```

```
1  $ python myscript.py
2  The DNA sequence is empty
3  +++ sequence analysed +++
4  $
```

Lesson: if-body is executed if expression evaluates to True

Control Flow: if Statement

myscript.py

```
if False:
    print("a")
    print("b")

if True:
    print("c")
    print("d")

print("e")
```

```
$ python myscript.py
```

Control Flow: if Statement

myscript.py

```
1  if False:
2      print("a")
3      print("b")
4
5  if True:
6      print("c")
7      print("d")
8
9  print("e")
```

```
1  $ python myscript.py
2  c
3  d
4  e
5  $
```

Lessons:

- All statements on the same indentation level form a **block**
- In control flow statements a block is often called the **body** of the statement
- A block (body) is entirely executed

Control Flow: if Statement

myscript.py

```
1  seq = "ACTGGAGTCAGG"
2  seq_len = len(seq)
3
4  if seq_len == 0:
5      print("The DNA sequence is empty")
6  else:
7      a_content = seq.count("A")
8      print("Adenin content: " + str( a_content ))
9
10 print("+++ sequence analysed +++")
```

```
1  $ python myscript.py
```

Control Flow: if Statement

myscript.py

```
1  seq = "ACTGGAGTCAGG"
2  seq_len = len(seq)
3
4  if seq_len == 0:
5      print("The DNA sequence is empty")
6  else:
7      a_content = seq.count("A")
8      print("Adenin content: " + str( a_content ))
9
10 print("+++ sequence analysed +++")
```

```
1  $ python myscript.py
2  Adenin content: 3
3  +++ sequence analysed +++
4  $
```

Lesson: else-body is executed if expression evaluates to False

Control Flow: if Statement

myscript.py

```
seq = "ACTGGAGTCAGG"
seq_len = len(seq)

if seq_len == 0:
    print("The DNA sequence is empty")
else:
    a_content = seq.count("A")
    a_freq = seq.count("A")
    print("Adenin  frequency: " + str( seq.count("A") / seq_len ))
print("+++ sequence analysed +++")
```

```
$ python myscript.py
```

Control Flow: if Statement

myscript.py

```
1  seq = "ACTGGAGTCAGG"
2  seq_len = len(seq)
3
4  if seq_len == 0:
5      print("The DNA sequence is empty")
6  else:
7      a_content = seq.count("A")
8      a_freq = seq.count("A")
9      print("Adenin  frequency: " + str( seq.count("A") / seq_len ))
10 print("+++ sequence analysed +++")
```

```
1  $ python myscript.py
2  Adenin  frequency: 0.25
3  +++ sequence analysed +++
4  $
```

Lesson: if statements can be used to prevent potential errors

Control Flow: if Statement

myscript.py

```
1  age = 43
2
3  if age <= 6: print("Ticket price: free")
4  elif age <= 18:
5      print("Ticket price: 75 € (25% discount)")
6  elif age <= 65:
7      print("Ticket price: 100 € ")
8  else:
9      print("Sure you want to visit a Tokio Hotel concert?")
10
11 print("... proceed to checkout ...")
```

```
1  $ python myscript.py
```

Control Flow: if Statement

myscript.py

```
1  age = 43
2
3  if age <= 6: print("Ticket price: free")
4  elif age <= 18:
5      print("Ticket price: 75 € (25% discount)")
6  elif age <= 65:
7      print("Ticket price: 100 € ")
8  else:
9      print("Sure you want to visit a Tokio Hotel concert?")
10
11 print("... proceed to checkout ...")
```

```
1  $ python myscript.py
2  Ticket price: 100 €
3  ... proceed to checkout ...
4  $
```

Lessons:

- elif-statements are additional if statements that are sequentially evaluated
- The first if or elif expression that evaluates to True is executed, the rest is skipped

Control Flow: for Loop

- A **for loop** allows repeated execution of its body
- The number of repetitions is known: so-called definite iteration
- A for loop walks through a so-called **iterable** (e.g. a sequence)
- The current element is assigned to a **loop variable**

myscript.py

```
1  lst = range(5)
2
3  for n in lst:
4      print(n)
```

```
1  $ python myscript.py
```

Control Flow: for Loop

- A **for loop** allows repeated execution of its body
- The number of repetitions is known: so-called definite iteration
- A for loop walks through a so-called **iterable** (e.g. a sequence)
- The current element is assigned to a **loop variable**

myscript.py

```
1  lst = range(5)
2
3  for n in lst:
4      print(n)
```

```
1  $ python myscript.py
2  0
3  1
4  2
5  3
6  4
7  $
```

Control Flow: for Loop

myscript.py

```
lst = ["Harry", "Brian", "Bob", "Jimmy", "Vincent", "Brad"]

for name in lst:
    if name == "Bob":
        break
    print(name)
```

```
$ python myscript.py
```

Control Flow: for Loop

myscript.py

```
1  lst = ["Harry", "Brian", "Bob", "Jimmy", "Vincent", "Brad"]
2
3  for name in lst:
4      if name == "Bob":
5          break
6      print(name)
```

```
1  $ python myscript.py
2  Harry
3  Brian
4  $
```

Lessons:

- the behaviour of the for loop can be influenced
- the keyword **break** terminates the loop
- control flow statements can be **nested**

Control Flow: for Loop

myscript.py

```
lst = ["Harry", "Brian", "Bob", "Jimmy", "Vincent", "Brad"]

for name in lst:
    if name[0] != "B":
        continue
    print(name)
```

```
$ python myscript.py
```

Control Flow: for Loop

myscript.py

```
1  lst = ["Harry", "Brian", "Bob", "Jimmy", "Vincent", "Brad"]
2
3  for name in lst:
4      if name[0] != "B":
5          continue
6      print(name)
```

```
1  $ python myscript.py
2  Brian
3  Bob
4  Brad
5  $
```

Lessons:

- the behaviour of the for loop can be influenced
- the keyword **continue** terminates the current iteration

Control Flow: while Loop

- A **while loop** allows repeated execution of its enclosed body
- The number of repetitions is variable: so-called indefinite iteration
- A while loop evaluates a boolean expression
- **True**: the body of the while loop is executed
- **False**: loop terminates and first statement after it is executed

myscript.py

```
1  n = 0
2
3  while n < 100:
4      print(str(n) + " ", end='')
5      n += 1
6
7  print()
```

```
1  $ python myscript.py
```

Control Flow: while Loop

- A **while loop** allows repeated execution of its enclosed body
- The number of repetitions is variable: so-called indefinite iteration
- A while loop evaluates a boolean expression
- **True**: the body of the while loop is executed
- **False**: loop terminates and first statement after it is executed

myscript.py

```
1  n = 0
2
3  while n < 100:
4      print(str(n) + " ", end='')
5      n += 1
6
7  print()
```

```
1  $ python myscript.py
2  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
3  35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
4  67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
5  $
```

Control Flow: Functions

- Functions perform specific tasks: e.g. len(), print(), ...
- Besides the built-in functions, arbitrary functions can be defined
- Functions allow to structure complex code
- Functions allow to reduce redundancy and to write reusable code

- A function needs at least a **name**, a **parameter list** , and a **body**
- The name follows the naming rules of variables
- The parameter list allows to pass parameters and may be empty
- The body contains the statements required to fulfill the desired task
- Functions are identified by the keyword **def**

Control Flow: Functions

myscript.py

```
1  def say_hello():
2      print("Hello")
```

```
1  $ python myscript.py
```

- A function definition starts with keyword **def**
- Next comes the **function name**: here: 'say_hello'
- Next comes the **parameter list** in parentheses: here empty
- And don't forget the **':'**
- The body of the function is **indented**

Control Flow: Functions

myscript.py

```
1  def say_hello():
2      print("Hello")
```

```
1  $ python myscript.py
2  $
```

Lessons:

- Nothing happens
- Functions need to be **called** to be executed

Control Flow: Functions

myscript.py

```
def say_hello():
    print("Hello")

say_hello()
```

```
$ python myscript.py
```

Control Flow: Functions

myscript.py

```
1  def say_hello():
2      print("Hello")
3
4  say_hello()
```

```
1  $ python myscript.py
2  Hello
3  $
```

Control Flow: Functions

myscript.py

```
def repeat(par_1):
    print(":: " + par_1)

var_1 = input()
while True:
    if var_1 == "":
        break
    repeat(var_1)
    var_1 = input()
```

```
$ python myscript.py
```

Control Flow: Functions

myscript.py

```
def repeat(par_1):
    print(":: " + par_1)

var_1 = input()
while True:
    if var_1 == "":
        break
    repeat(var_1)
    var_1 = input()
```

```
$ python myscript.py
Hello
:: Hello
Parrot
:: Parrot

$
```

Lesson: parameters (here: par_1) can be used to pass data to the function

Control Flow: Functions

myscript.py

```
def repeat(par_1, par_2):
    print(par_2 + par_1)

var_1 = input()
while True:
    if var_1 == "":
        break
    repeat(var_1, "++ ")
    var_1 = input()
```

```
$ python myscript.py
```

Control Flow: Functions

myscript.py

```
1  def repeat(par_1, par_2):
2      print(par_2 + par_1)
3
4  var_1 = input()
5  while True:
6      if var_1 == "":
7          break
8      repeat(var_1, "++ ")
9      var_1 = input()
```

```
1  $ python myscript.py
2  Hello
3  ++ Hello
4  Parrot
5  ++ Parrot
6
7  $
```

Lesson: multiple parameters are comma-separated

Control Flow: Functions

- Functions can return something: e.g. a value or arbitrary object
- The **return** statement can be used for that purpose

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      print("-- calculation done --")
4      return res
5
6  result = adder(1, 2)
7  print(result)
8  print( adder(1, 2) )
```

```
1  $ python myscript.py
```

Control Flow: Functions

- Functions can return something: e.g. a value or arbitrary object
- The **return** statement can be used for that purpose

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      print("-- calculation done --")
4      return res
5
6  result = adder(1, 2)
7  print(result)
8  print( adder(1, 2) )
```

```
1  $ python myscript.py
2  -- calculation done --
3  3
4  -- calculation done --
5  3
6  $
```

Lesson: the return statement can pass back data to the caller

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      return res
4      print("-- calculation done --")
5
6  result = adder(1, 2)
7  print(result)
8  print( adder(1, 2) )
```

```
1  $ python myscript.py
```

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      return res
4      print("-- calculation done --")
5
6  result = adder(1, 2)
7  print(result)
8  print( adder(1, 2) )
```

```
1  $ python myscript.py
2  3
3  3
4  $
```

Lesson: the return statement terminates the function immediately

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      return
4      print("-- calculation done --")
5
6  result = adder(1, 2)
7  print(result)
8  print( type(result) )
```

```
1  $ python myscript.py
```

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      return
4      print("-- calculation done --")
5
6  result = adder(1, 2)
7  print(result)
8  print( type(result) )
```

```
1  $ python myscript.py
2  None
3  <class 'NoneType'>
4  $
```

Lessons:

- The return statement can be without argument
- I this case, the return value is 'None' with data type NoneType
- Indeed, a function without return statement returns 'None' anyways

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      return
4      print("-- calculation done --")
5
6  result = adder(1, 2)
7  print(result)
8  print( type(result) )
```

```
1  $ python myscript.py
```

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2      res = s1 + s2
3      return
4      print("-- calculation done --")
5
6  result = adder(1, 2)
7  print(result)
8  print( type(result) )
```

```
1  $ python myscript.py
2  None
3  <class 'NoneType'>
4  $
```

Lessons:

- The return statement can be without argument
- I this case, the return value is 'None' with data type NoneType
- Indeed, a function without return statement returns 'None' anyways

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2
3      def calc(s1, s2):
4          res = s1 + s2
5          return res
6
7      res = calc(s1, s2)
8      return res
9
10 print( adder(1,2) )
```

```
1  $ python myscript.py
```

Control Flow: Functions

myscript.py

```
1  def adder(s1, s2):
2
3      def calc(s1, s2):
4          res = s1 + s2
5          return res
6
7      res = calc(s1, s2)
8      return res
9
10 print( adder(1,2) )
```

```
1  $ python myscript.py
2  3
3  $
```

Lesson: also functions can be **nested**

Scope

- The concept of **scopes** is important in many programming languages
- Scopes determine in which places **identifiers** are **visible**
- Identifiers are names of variables, functions, or classes
- The scope of an identifier depends on the place of its definition
- When talking about scopes in Python we talk about (nested) functions and the top-level script (module)
- Python works with four scopes:
  local, **enclosed**, **global**, **built-in** (**LEGB Rule**)
- To find an identifier Python looks in these scopes in the given order

Scope

- The concept of **scopes** is important in many programming languages
- Scopes determine in which places **identifiers** are **visible**
- Identifiers are names of variables, functions, or classes
- The scope of an identifier depends on the place of its definition
- When talking about scopes in Python we talk about (nested) functions and the top-level script (module)
- Python works with four scopes:
  local, **enclosed**, **global**, **built-in** (**LEGB Rule**)
- To find an identifier Python looks in these scopes in the given order
- Let's do some examples :)

Scope

myscript.py

```
def fun1():
    # Local scope of function fun1()
    var1 = 100
    print( var1 )

fun1()
print( var1 )
```

```
$ python myscript.py
```

Scope

myscript.py

```
1  def fun1():
2      # Local scope of function fun1()
3      var1 = 100
4      print( var1 )
5
6  fun1()
7  print( var1 )
```

```
1  $ python myscript.py
2  100
3  Traceback (most recent call last):
4    File "myscript.py", line 6, in <module>
5      print( var1 )
6  NameError: name 'var1' is not defined
7  $
```

Lessons:

- The **local scope** of a name is the body of the function in which it is defined
- A name is visible in the scope where it is defined
- In outer scopes (outer functions, less indentation) the name is not visible

Scope

myscript.py

```
def fun1():
    # Enclosing scope
    var1 = 100

    def fun2():
        # Enclosed scope: names from enclosing scopes are visible
        var2 = 200
        print(var1)

    fun2()
fun1()
```

```
$ python myscript.py
```

Scope

myscript.py

```
1  def fun1():
2      # Enclosing scope
3      var1 = 100
4
5      def fun2():
6          # Enclosed scope: names from enclosing scopes are visible
7          var2 = 200
8          print(var1)
9
10     fun2()
11 fun1()
```

```
1  $ python myscript.py
2  100
3  $
```

Lessons:

- Names defined in an **enclosing scope** (outer) are visible in their enclosed (inner) scopes
- Enclosing scopes contain nested functions, which form enclosed scopes

Scope

myscript.py

```
def fun1():
    def fun2():
        print(var1)

    fun2()
    var1 = 100

fun1()
```

```
$ python myscript.py
```

Scope

myscript.py

```
1  def fun1():
2      def fun2():
3          print(var1)
4
5      fun2()
6      var1 = 100
7
8  fun1()
```

```
1  $ python myscript.py
2  Traceback (most recent call last):
3    File "myscript.py", line 8, in <module>
4      fun1()
5    File "myscript.py", line 5, in fun1
6      fun2()
7    File "myscript.py", line 3, in fun2
8      print(var1)
9  NameError: free variable 'var1' referenced before assignment in enclosing scope
10 $
```

Lessons:

- Always: a variable needs also to be assigned before it is used

Scope

myscript.py

```
var0 = 300

def fun1():
    print(var0)
    def fun2():
        print(var0)

    fun2()
fun1()
```

```
$ python myscript.py
```

Scope

myscript.py

```
var0 = 300

def fun1():
    print(var0)
    def fun2():
        print(var0)

    fun2()
fun1()
```

```
$ python myscript.py
300
300
$
```

Lessons:

- The **global scope** is the top-level script (module)
- Names defined on the global scope are visible from any place
- We can consider global scope as the 'all-enclosing-scope'

Scope

- We used **visibility** of names over different scopes so far
- Visibility does not mean names from other scopes can be modified

myscript.py

```
1  var1 = 100
2
3  def fun1():
4      var1 = 200
5      print(var1)
6
7  fun1()
8  print(var1)
```

```
1  $ python myscript.py
```

Scope

- We used **visibility** of names over different scopes so far
- Visibility does not mean names from other scopes can be modified

myscript.py

```
1  var1 = 100
2
3  def fun1():
4      var1 = 200
5      print(var1)
6
7  fun1()
8  print(var1)
```

```
1  $ python myscript.py
2  200
3  100
4  $
```

Lessons:

- assignment in local space creates a new name in that scope
- the enclosed name **shadows** the equivalent name in the enclosing scope

Scope

myscript.py

```
var1 = 100

def fun1():
    var1 = var1 + 1
    print(var1)

fun1()
```

```
$ python myscript.py
```

Scope

myscript.py

```
1  var1 = 100
2
3  def fun1():
4      var1 = var1 + 1
5      print(var1)
6
7  fun1()
```

```
1  $ python myscript.py
2  Traceback (most recent call last):
3    File "test.py", line 7, in <module>
4      fun1()
5    File "test.py", line 4, in fun1
6      var1 = var1 + 1
7  UnboundLocalError: local variable 'var1' referenced before assignment
8  $
```

Lessons:

- You even cannot update (change) names from enclosing scopes
- Names from outer scopes are read-only
- This behaviour can be changed but isn't recommended

Diving Deeper ...

Control Flow

→ https://www.w3schools.com/python/python_for_loops.asp

→ https://realpython.com/python-conditional-statements/

→ https://realpython.com/python-for-loop/

→ https://docs.python.org/3/tutorial/controlflow.html

Functions

→ https://www.w3schools.com/python/python_functions.asp

→ https://realpython.com/defining-your-own-python-function/

→ https://docs.python.org/3/tutorial/controlflow.html#defining-functions

Scope

→ https://www.w3schools.com/python/python_scope.asp

→ https://realpython.com/python-scope-legb-rule/

Practice Time ... 004

1. p001: Write a script that keeps reading user input until the input is empty.

2. p002: Extend your script and inform user if input string is a valid integer number.
   If the input is not a valid number please also print an appropriate information.
   Hint: do some research on the available methods of the class (data type) string.

3. p003: Extend your script and inform user if the validated numeric input is even or odd.

4. p004: Write a script that reads 3-letter amino acid code and prints the full amino acid name.
   Please feel free to implement just a subset of the 20 proteinogenic amino acids.

5. p005: Extend your 'amino-acid-speller' to be case insensitive.
   That is, it accepts 'ala' as well as 'ALA' as well as 'alA' ... as valid inputs.
   Hint: do again some research on the available str methods.

License and Contributors

Main Author
Philipp Thiel

Additional Contributors
n.a.