Programming Basics

Editor: Martijn Stegeman

January 29, 2021

Chapter 1

Functions

1.1 Functions that print

Write down what is printed when running each of the following code fragments.

1.1

```
def baz():
    print("fly")
baz()
```

1.2

```
def bar():
    print("jump")
```

1.3

```
def foo():
    print("fly")
def bar():
    print("jump")
foo()
bar()
bar()
```

1.4

```
def baz():
    print("jump")
def foo():
    print("bicycle")
baz()
```

1.5

```
def foo():
    print("pie")
def baz():
    foo()
    print("bicycle")
foo()
baz()
```

1.6

```
def baz():
    bar()
    print("rainbow")
def bar():
    print("fly")
baz()
```

1.7

```
def foo():
    baz()
    print("cake")
def baz():
    print("jump")
baz()
```

1.8

```
def bar():
    print("bicycle")
def baz():
    print("jump")
    bar()
baz()
```

Defining a function There is a separation between the fuction *definition*, which describes what actions the function will perform, and the function *call*, which signals that our function should be run. This means that as we define a function, it will not automatically be run, allowing us to postpone running it until we need it. This, in turn, allows us to call a function multiple times in different parts of our programs. In this book, a function definition will look like this:

```
def <name of function>():
    <function body>
```

The word `def` signals that we are about to define a function. For now, we will just discuss functions that perform actions, we will see functions that calculate something later. One example of a function that performs an action is a printing function:

```
def print_reassuring_message():
    print("I'm still here!")
```

Calling a function Calling a function can be done from anywhere in the program as long as it has been *defined* previously. Calling a function looks like this:

```
<name of function>()
```

As you can see, the parentheses `()` are an important part of the function definition, as well as of the function call. Most programming languages use these to discern functions from other elements of the program.

Tracing Tracing is a method for manually simulating the execution of code, in order to verify that all steps work as expected. Function calls can be traced, but because the definition and calls to functions are separate, we need a clear notation. Say we take the program below. The execution of the program starts at the first line that is not a function definition (in our case, the last line). On this line, the function `baz` is called. In the function `baz`, `bar` is called, which we can then "jump" to. The key is to strictly follow the top-down sequence of statements, unless there is a function call.

```
  def  baz():
2  bar()
4  print("rainbow")
  def  bar():
3  print("fly")
1 baz()
```

In our trace we draw lines next to the functions to have a clear separation, and we number the lines in order of execution. We see that "fly" is printed before "rainbow".

1.2 Parameters

Write down what is printed when running each of the following code fragments.

1.9

```
def hay(x):
   print(x)
hay("fly")
```

1.10

```
def dal(y):
   print(y)
dal("jump")
```

1.11

```
def eau(x, y):
   print(x / y)
eau(10, 12)
```

1.12

```
def pow(y, x):
   print(x / y)
pow(10, 12)
```

1.13

```
def gam(x, z):
   print(z / x)
gam(10, 12)
```

1.14

```
def zek(x, z):
   print(x / z)
zek(12, 10)
```

1.15

```
def eid(x, y, z):
   print(z / x)
eid(10, 11, 12)
```

1.16

```
def ash(y, x, z):
   print(x / z)
ash(12, 11, 10)
```

1.17

```
def bar(z, y, x):
   print(y / x * z)
bar(10, 11, 12)
```

1.18

```
def duo(x, z, y):
   print(y / z)
duo(12, 11, 10)
```

1.19

```
def oca(y, z, x):
   print(x / y)
oca(10, 11, 12)
```

1.20

```
def tug(z, x, y):
   print(x / z * y)
tug(12, 11, 10)
```

Parameters Functions often have *parameters*. When calling the function, we supply *values* for these parameters, which is called *parameter passing*. From the function's perspective, these values are assigned names that are specified in the *parameter list* of the function definition.

```
def <name of function>(<parameter list>):
    <function body>
```

Now consider these two function definitions. Both have two parameters named in their respective parameter lists.

```
def date_1(day, month):
    print(day)
    print(month)
```

```
def date_2(month, day):
    print(day)
    print(month)
```

In the left definition we specify the parameters `day` and `month`. In the right definition, we specify `month` and `day`, in reverse order. This *order* has an effect on what names are given to the values that are passed when calling the function. Let's call the functions:

```
date_1(21, 6)
```

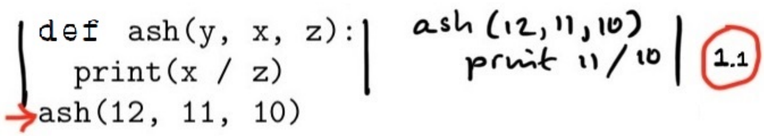
```
date_2(6, 21)
```

The output of the functions would then be:

```
21
6
```

```
21
6
```

Tracing Keeping track of all values when passing parameters can easily become very tedious, which is why we often need to trace them explicitly. Below, like before, we put a line next to the one function that is defined. We also mark the starting line with a little arrow.



On that starting line, the function `ash` is called. To the right of the definition of that function, we draw a line, and copy the function definition, while substituting the concrete values from the *function call*.

Combining the original function definition and the substituted version, we can infer that in the function `y = 12`, `x = 11` and `z = 10`. We use this information to substitute the values of `x` and `z` on the line containing `print`.

Then, only one calculation is left, the result of which will be printed. When we evaluate the expression, we get the number that will be printed: `1.1`.

1.3 Calling functions with variables

Write down what is printed when running each of the following code fragments.

1.21

```
def foo(x):
   print(x)
inv = "fly"
foo(inv)
```

1.22

```
def bar(name):
   print(name)
foo = "skip"
bar("jump")
```

1.23

```
var = "bear"
def una(var):
   print(var)
una(var)
```

1.24

```
name = "pieter"
def greet(name):
   print("hello")
greet(name)
```

1.25

```
x = 3
y = 5
def foo(a, b):
   print(a / b)
foo(x, y)
```

1.26

```
x = 3
y = 5
def baz(y, x):
   print(x / y)
baz(x, y)
```

1.27

```
x = 10
def bar(x, z):
   print(z / x)
bar(x, 12)
```

1.28

```
var1 = 2
def dra(var1, var2):
   print(var1 + var2)
dra(var1, 8)
```

1.29

```
x = "fly"
var2 = "skip"
def shout(var1, var2):
   print(var1 + var2)
shout(x, "jump")
```

1.30

```
x = 3
a = 2
def magic(a, b):
   print(a - b)
magic(x, a)
```

Passing variables Functions can also accept the values from variables as concrete values for their parameters. Let's study the following function:

```
def twice(text):
    print(text)
    print(text)
fruits = "pineapple blueberry"
twice(fruits)
```

```
pineapple blueberry
pineapple blueberry
```

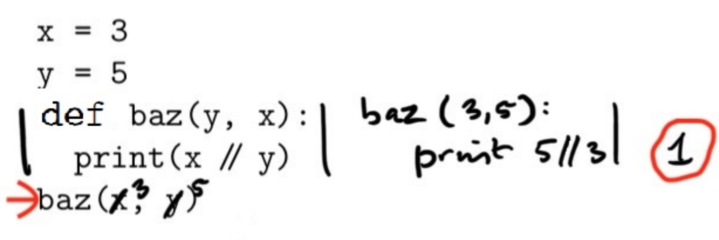
We passed a single variable called `fruits` to the function. The value of the variable `fruits` is passed to the function, where it will now be named `text`. This `text` is then printed to the screen twice.

Like before, the order of the values that are passed to a function will determine the parameter names these values get. This is *always* done in order. This is especially important, because sometimes parameters may have the same name as variables found elsewhere in the code. Consider this fragment:

```
x = 10
y = 40
def minus(x, y):
    print(x - y)
minus(y, x)
```

Because the values of `y` and `x` are passed to the function in that order, inside the function we will have the parameters and their respective values `x = 40` and `y = 10`. Hence, the result that is printed will be 30.

Tracing Because of the potential confusion between variable and parameter names, we add an explicit step to our function tracing technique: substituting values in the function call. We cross out the variable name and write its value next to it. We can do this before considering the function's definition at all. Now that we have substituted the concrete values, it's easy to copy them into the function call like in earlier sections.



Chapter 2

Functions that calculate

2.1 Functions that return something

Write down what is printed when running each of the following code fragments.

2.1

```
def foo(x):
   return x
print(foo("hard"))
```

2.2

```
def bar(name):
   return "jump"
print(bar("hi"))
```

2.3

```
def foo(x, y):
   return x / y
print(foo(10, 12))
```

2.4

```
def baz(y, x):
   return x / y
print(baz(10, 12))
```

2.5

```
amd = 4
def oei(dam, mad):
   return dam * mad + amd
sim = oei(amd, amd)
print(sim)
```

2.6

```
x = 1
def una(x):
   return x + 1
def zab(y):
   return y - 1
print(zab(2) + una(x))
```

2.7

```
def mak(x, y, z):
   return y
mis = "cal"
res = mak("sol", mis, "toa")
print(res)
```

2.8

```
def una(x, z):
   print(x / z)
def kam(y):
   return y * 3
print(una(kam(4), 10))
```

Returning To *return* means to transfer back some value from the function to where it was called. Most programming languages use the keyword *return* for this. The value that is returned is sometimes called the *result* of the function. In the code where the function is called, the function call is replaced by its result in the same way that we use replace variable names with their respective values each time we calculate using variables.

Tracing Functions that calculate and return something can be traced much like before. We add a *back substitution* that fills in the returned value of the function into the place where it was called. Below, the function call `das(12)` returns the float 3.0, which is then substituted into the `print` statement.

```
| def das(x):         | das(12):
|     return x / 4    |   return 12/4 | 3.0
→print(das(12))       | print(3.0)
```

2.2 Functies that call other functions

Write down what is printed when running each of the following code fragments.

2.9

```
def gus(z, y)
   return dim(z, y) / 2
def yap(z, x)
   return gus(x, z) - 3
def dim(y, x)
   return x + 4
x = 6
y = 4
print(yap(x, y))
```

2.10

```
def uru(x, y)
   return 2 * y
def sal(z, y)
   return 4 / uru(y, z)
def hit(z, x)
   return 2 + sal(z, x)
y = 5
x = 6
print(hit(x, y))
```

2.11

```
def dip(y, x)
   return y * 4
def mux(y, z)
   return 5 / dip(y, z)
def bad(x, z)
   return mux(z, x) + 3
y = 4
x = 5
print(bad(x, y))
```

2.12

```
def loo(y, z)
   return z / 3
def aam(z, y)
   return loo(y, z) * 6
def wah(y, z, x)
   return aam(y, z) - 2
z = 2
x = 5
y = 4
print(wah(x, y, z))
```

2.13

```
def nae(z, y)
   return 2 * y
def bus(x, z)
   return nae(z, x) + 3
def ann(z, y, x)
   return bus(y, z) / 2

y = 5
x = 3
z = 6
print(ann(x, y, z))
```

Stacked function calls Functions that calculate and return something can also call other functions that return something. Tracing might look like below. Using red digits, we indicate the order in which the trace is done.

```
def fli(x):
    return 2 + x
def fla(y):
    return 10 - fli(y)
def flo(z):
    return 2 * fla(z)
print(flo(5))
```

4 fli(5): return 2+5	5 7	
3 fla(5): return 10-fli(5)	6 10-7	7 3
2 flo(5): return 2* fla(5)	8 2*3	9 6
1 → print(flo(5))	10 print(6)	

Starting from the `print`, we see that the function `flo` is called with the value `5`. The function `flo` returns two times the result of the function `fla`, which is called with the variable `z` that has the value `5`. We can see that the function `fla` returns ten minus the result of the function `fli`, with the variable `y`, which has the value `5`. The function `fli` returns two plus the variable `x`, which has the value `5`. The outcome of `2 + 5 = 7`, so we can use back substitution to replace `fli(5)` with `7`. Now, we can calculate `10 - 7 = 3`, which means we can then replace `fla(7)` with `3`. Finally, we can calculate `2 * 3 = 6`, which is then substituted into the print.

Answers

Exercises

1.1	1.2	1.3	1.4
`fly`		`fly jump jump`	`jump`

1.5	1.6	1.7	1.8
`pie pie bicycle`	`fly rainbow`	`jump`	`jump bicycle`

1.9	1.10	1.11	1.12	1.13	1.14
`fly`	`jump`	`0`	`1`	`1`	`1`

1.15	1.16	1.17	1.18	1.19	1.20
`1`	`1`	`0`	`0`	`1`	`0`

1.21	1.22	1.23	1.24	1.25	1.26
`fly`	`jump`	`bear`	`hello`	`0`	`1`

1.27	1.28	1.29	1.30
`1`	`10`	`fly jump`	`1`

2.1	2.2	2.3	2.4	2.5	2.6
`hard`	`jump`	`0`	`1`	`20`	`3`

2.7	2.8
`cal`	`1`

2.9	2.10	2.11	2.12	2.13
2	2	3	4	6