15-110 Principles of Computing – S19

Lecture 10: Functions 1

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Functions: callable, named subprograms (procedures)

- **Function:** informally, a *subprogram,* we write a sequence of statements and give that sequence a name. The instructions can then be executed at any point in a program by referring to the function name

```
def function_name(arguments):
    function_body
    return something
```

User function definition

```
def happy():
    print("Happy birthday to you!")

name = 'Fred'
print('Hello ' + name + '!')
happy()
happy()
happy()
```

Calling (invoking) the function

Built-in function (**python standard library**)

```
def function_name(arguments):
    function_body
```

is implemented as:

```
def function_name(arguments):
    function_body
    return None
```

All function calls return something, None when left unspecified in the function body

Built-in functions (Python standard library)

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

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

Functions: organizing the code, putting aside functionalities

- Functions are a fundamental way to *organize* the code into **procedural elements** that can be *reused*
- Functions provide **structure and organization,** that facilitate:

Decomposition: example

Goal: design a simple *lottery system*, where the user inputs three integer numbers and the system checks if she/he has won the lottery or not!


```
value_list = []
n = 0
while n < 3:
    input_str = input('Input an integer number: ')
    if input_str.isdigit():
        v = int(input_str)
        value_list.append(v)
        n += 1
prod = 1
for v in value_list:
    prod *= v
if not (prod % 13):
    print("You won!!!")
else:
    print("You didn't win...")
```

Decomposition: example

get three numbers from user inputs and put them in a list

make the product of the numbers in the list

check the winning condition and output the message

```
value_list = []
n = 0
while n < 3:
    input_str = input('Input an integer number: ')
    if input_str.isdigit():
        v = int(input_str)
        value_list.append(v)
        n += 1
prod = 1
for v in value_list:
    prod *= v
if not (prod % 13):
    print("You won!!!")
else:
    print("You didn't win...")
```

- ✓ It can be observed that the overall task is (naturally) **composed by at least three independent *sub-tasks***
- ✓ In fact, the defined computation is composed by three groups of statements each aimed at realizing a specific sub-task → These three groups of statements can be conveniently *isolated*, organized into *different functions*, and then *merged* to obtain the same overall result
- ✓ Each function corresponds to the creation of a **primitive** for the task, that can be used in the computation

Decomposition: example

```
def get_user_numbers():
    value_list = []
    n = 0
    while n < 3:
        input_str = input('Input an integer number: ')
        if input_str.isdigit():
            v = int(input_str)
            value_list.append(v)
            n += 1
    return value_list

def make_product(values):
    prod = 1
    for v in values:
        prod *= v
    return prod

def check_lottery(input_val):
    if not (input_val % 13):
        print("You won!!!")
    else:
        print("You didn't win...")
```

```
numbers = get_user_numbers()
lottery_number = make_product(numbers)
check_lottery(lottery_number)
```

Decomposition: example

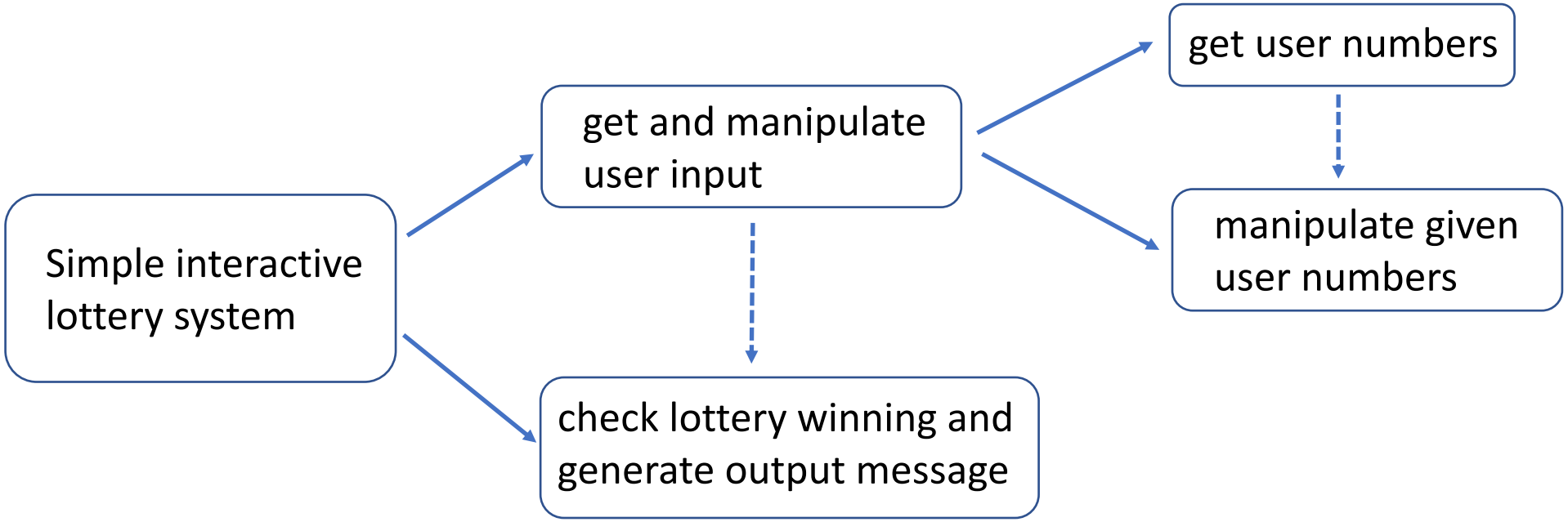
```
def get_user_numbers():
    value_list = []
    n = 0
    while n < 3:
        input_str = input('Input an integer number: ')
        if input_str.isdigit():
            v = int(input_str)
            value_list.append(v)
            n += 1
    return value_list

def make_product(values):
    prod = 1
    for v in values:
        prod *= v
    return prod

def check_lottery(input_val):
    if not (input_val % 13):
        print("You won!!!")
    else:
        print("You didn't win...")
```

```
numbers = get_user_numbers()
lottery_number = make_product(numbers)
check_lottery(lottery_number)
```

- Even better:

```
def get_lottery_number():
    values = get_user_numbers()
    number = make_product(values)
    return number

lottery_number = get_lottery_number()
check_lottery(lottery_number)
```

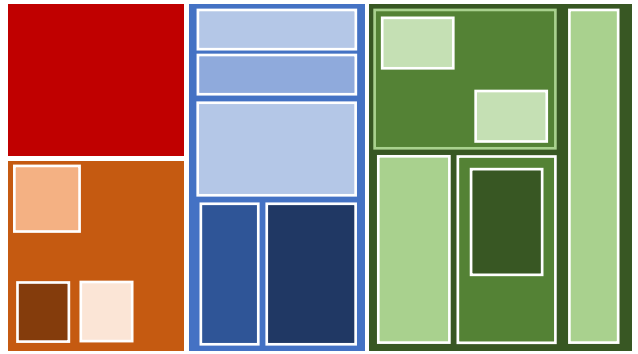
Decomposition using functions



Problem to solve (monolithic view)

Decompose in multiple (smaller / easier) sub-problems, possibly nested, and then merge the sub-problems

Divide-and-conquer



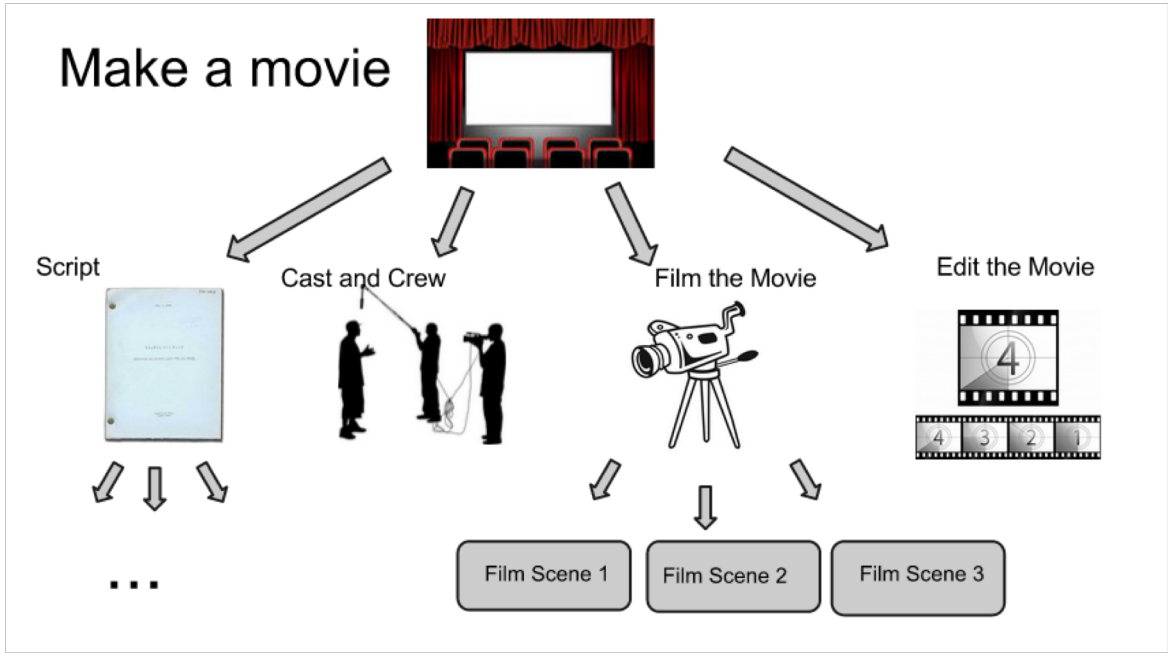
Gains from decomposition using functions

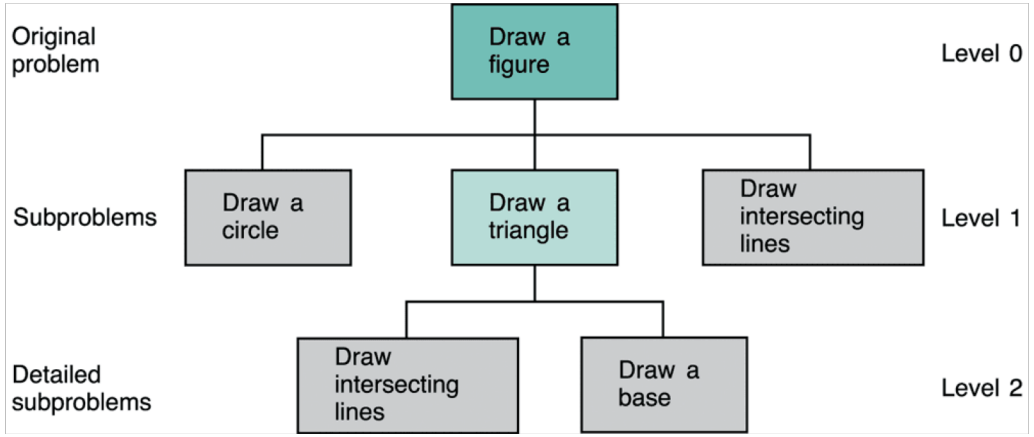
```
def get_user_numbers():
    value_list = []
    n = 0
    while n < 3:
        input_str = input('Input an integer number: ')
        if input_str.isdigit():
            v = int(input_str)
            value_list.append(v)
            n += 1
    return value_list

def make_product(values):
    prod = 1
    for v in values:
        prod *= v
    return prod
```

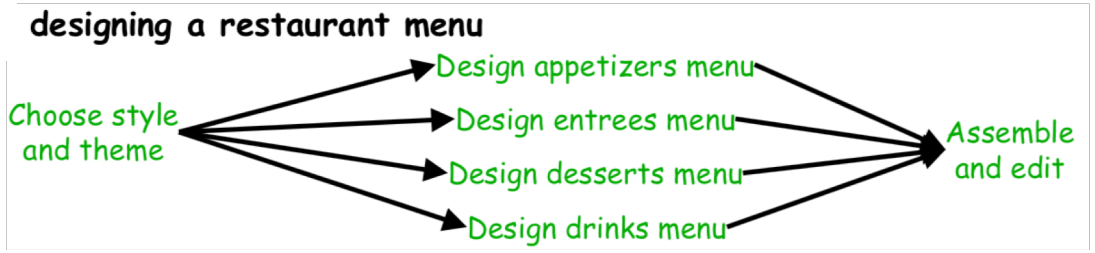
```
def get_lottery_number():
    values = get_user_numbers()
    number = make_product(values)
    return number

def check_lottery(input_val):
    if not (input_val % 13):
        print("You won!!!")
    else:
        print("You didn't win...")

lottery_number = get_lottery_number()
check_lottery(lottery_number)
```

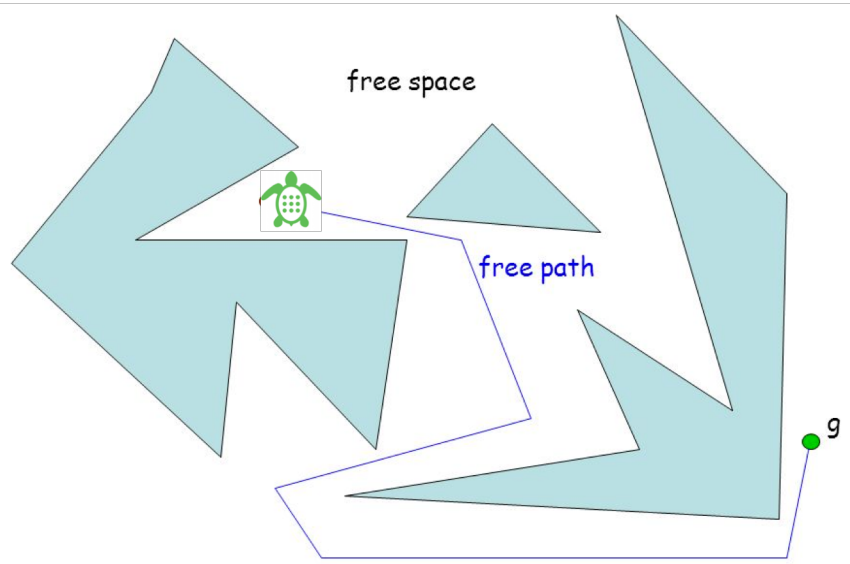
- Gaines resulting from decomposing the problem in functional blocks and defining functions / primitives to handle the computation in each block:
  - ✓ A better **understanding of the task** and of **our computational approach** to its solution
  - ✓ **Readability**: we can now *read* what's going on (using self-explanatory names for the functions)
  - ✓ **Easiness of testing**: each function can be tested for correctness / performance in standalone
  - ✓ **Easiness of maintenance and updating**: we can improve / change the code of each new primitive function independently from other program parts (as long as inputs and return types stay the same)
  - ✓ We can now even **split the job** to design the individual functions to multiple programmers*, in parallel!*

Examples of problem decompositions



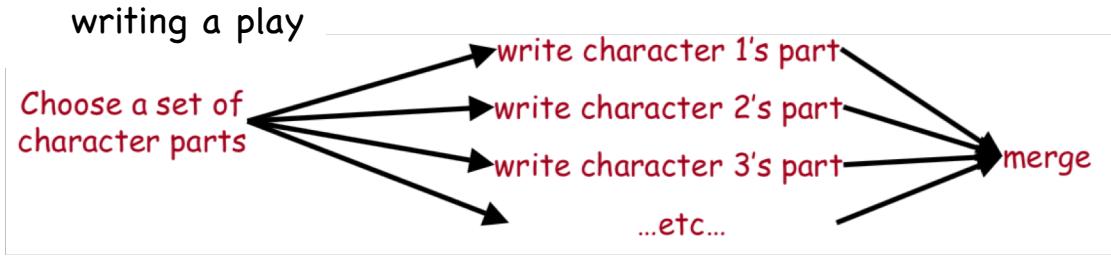






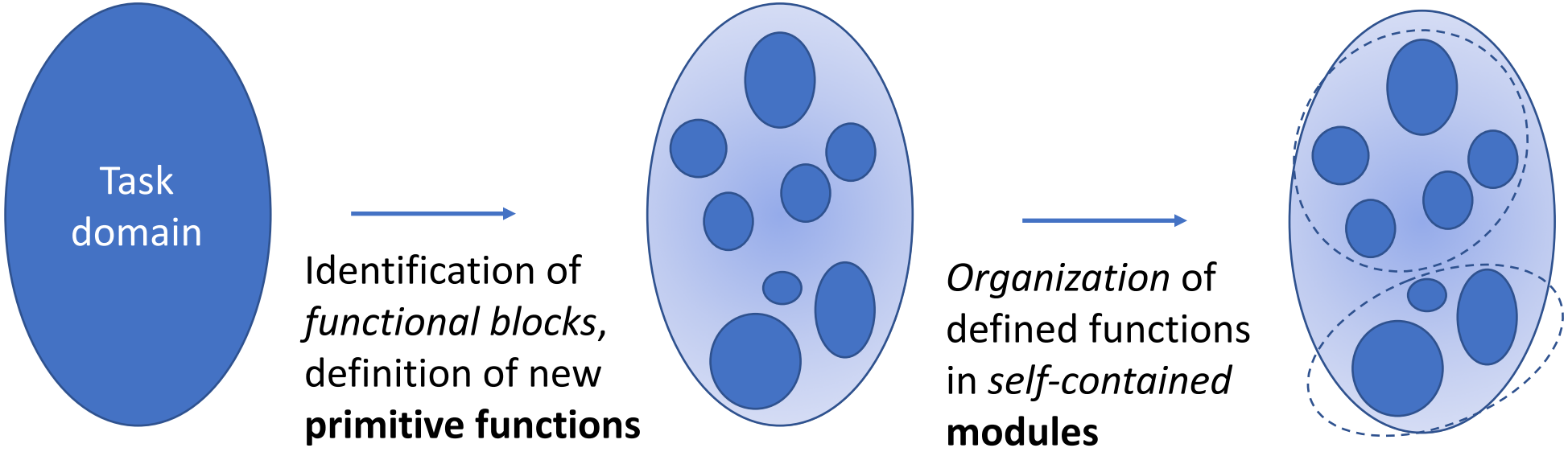
rotate in place $+110^\circ$
move forward 5m
rotate in place + $+80^\circ$
move forward 6m
rotate in place $+95^\circ$
move forward 7.5m
rotate in place -95°
move forward 2.5m
rotate in place -105°
move forward 9m
rotate in place -85°
move forward 2.8m

Not always fully feasible or straightforward!



Decomposition by creation of self-contained modules

- Decomposition can be realized at different granularity, depending on the task domain



A function is a module with one single component

Each module provides a set of (related) functionalities:

- ✓ **Reusability** in different tasks or different parts of the program
- ✓ High-level of **abstraction**

Reusability

- **Code reusability:** In addition to convenient code decomposition in functional blocks, the use of functions is essential to avoid to repeat the same fragment of code over and over in the same program, as well as to allow to reuse the same functional blocks in different programs

- E.g, we want to write a program that prints out the lyrics to the “Happy Birthday” song:

  Happy birthday to you!
  Happy birthday to you!
  Happy birthday, dear <insert-name>.
  Happy birthday to you!

- Without functions we would write something like the following code to wish happy birthday to Fred:

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear", 'Fred.')
print("Happy birthday to you!")
```

- If we want to wish happy birthday to Lucy too, now we have to duplicate the code … ☹

Reusability

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear", 'Fred.')
print("Happy birthday to you!")

print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear", 'Lucy.')
print("Happy birthday to you!")
```

We don't want all this duplicated code!

```
def happy():
    print("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy Birthday, dear", person + ".")
    happy()
```

- ✓ Now `sing(person)` can be reused for wishing happy birthday to *any* person!
- ✓ `sing()` and `happy()` could be included in a new `birthday` *module* …

Functions, parameters, scope, stack frames

```
def sing(person):
    happy()
    happy()
    print "Happy Birthday, dear", person + "."
    happy()
```

- ✓ The function defines an *abstraction*: it describes a computation that applies to any person (passed as a string), as defined through the input parameter `person`
- ✓ **Formal parameters** (***arguments***) play a central role to let a function being applied in different contexts
- ➢ Have the parameters local or global scope as variables?
- ➢ Do they live inside and/or outside of the function when the function is being called?

```
sing("Fred")
print(person)
```

Error!

Functions, parameters, scope, stack frames

What happens inside the computer when a function is called?

1. The **calling program *suspends*** at the point of the call and ***jumps*** to the program instruction with the definition of the function
2. A new **memory area** is allocated (stacked/*pushed* on the **run-time stack**) for the function (**call frame**)
3. The formal parameters of the function **get assigned the values supplied** by the actual parameters in the call, and get *pushed on the run-time stack* (**variable frame**)
4. The **body of the function is executed,** function **variables get pushed on the run-time stack**
5. At the end of the body or at return instruction, all call and variable frames are **popped out the stack**
6. Control returns **to the point just after where the function was called**

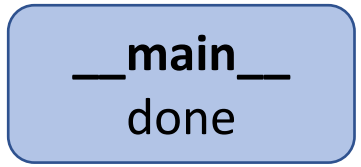
Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

- The **calling program *suspends*** at the point of the call and ***jumps*** to the program instruction with the definition of the function
- A new **memory area** is allocated (stacked/*pushed* on the **run-time stack**) for the function (**call frame**)
- The **body of the function is executed**
- Control returns **to the point just after where the function was called**

Functions and stack frames: no parameters/variables case

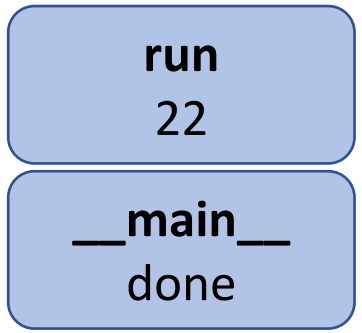
```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



Run-time stack

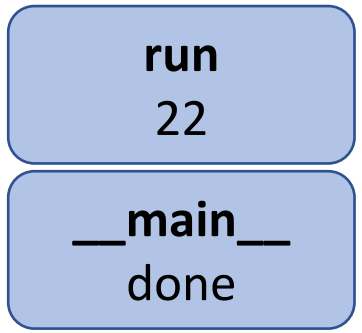
Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



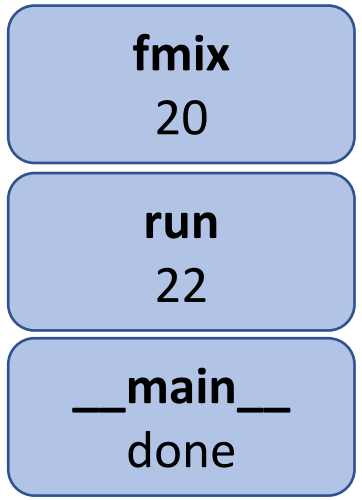
Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



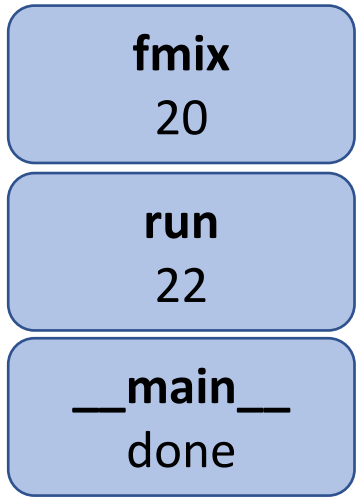
Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



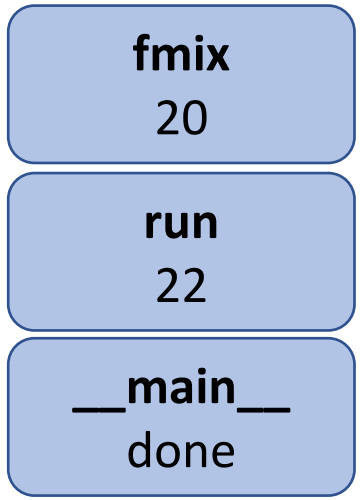
Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters /variables case

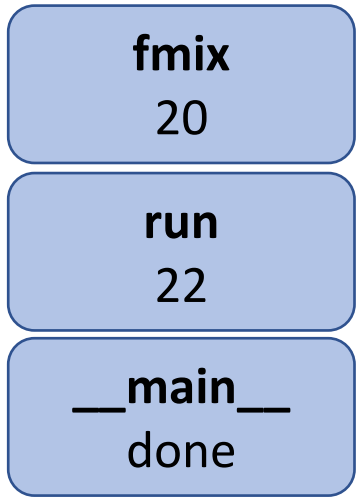
```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

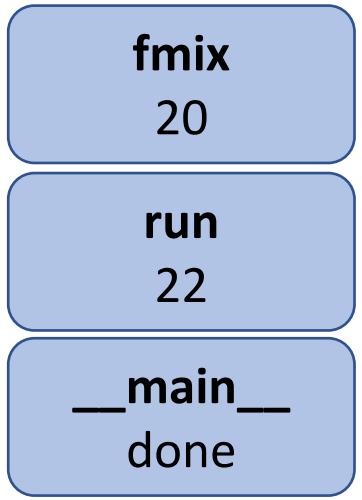
```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

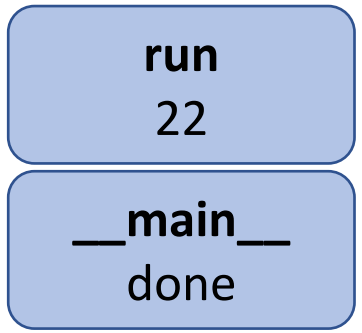
Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```



Functions and stack frames: no parameters/variables case

```
 1 def f1():
 2     print('f1 line 1')
 3     print('f1 line 2')
 4     print('f1 line 3')
 5     return
 6
 7 def f2():
 8     print('f2 line 1')
 9     print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

__main__
done

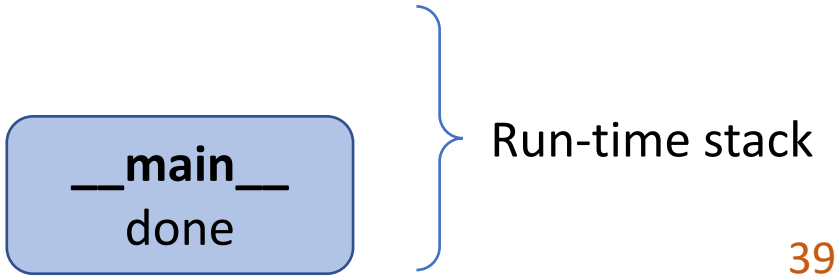
Functions and stack frames: no parameters/variables case

```
1  def f1():
2      print('f1 line 1')
3      print('f1 line 2')
4      print('f1 line 3')
5      return
6
7  def f2():
8      print('f2 line 1')
9      print('f2 line 2')
10     print('f2 line 3')
11     return
12
13 def fmix():
14     print('fmix line 1')
15     f2()
16     f1()
17     print('fmix line 4')
18
19 def run():
20     fmix()
21
22 run()
```

Run-time stack is empty!

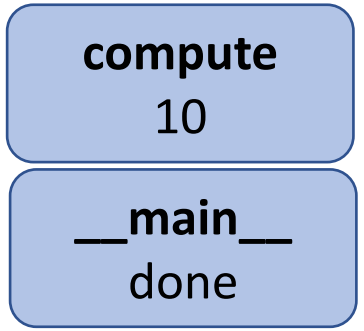
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



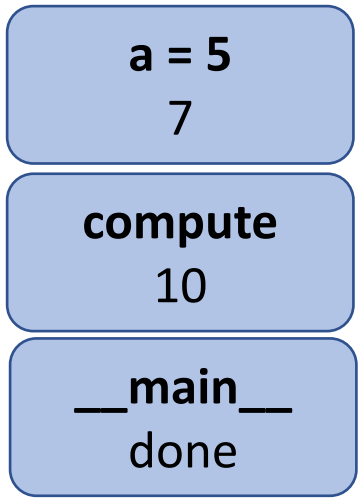
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



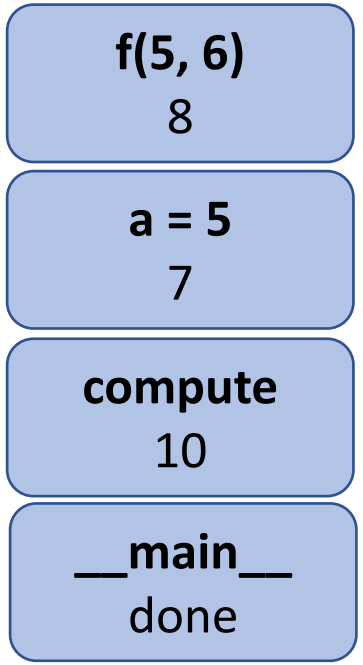
Functions and stack frames: with parameters and variables

```
1  def f(x, y):
2      print('x:', x, 'y:', y)
3      z = x + y
4      print('z:', z)
5
6  def compute():
7      a = 5
8      f(a, a + 1)
9
10 compute()
```



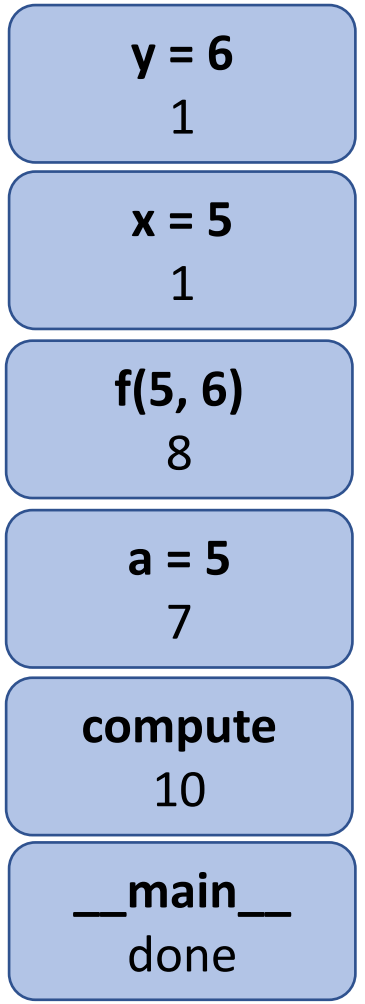
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



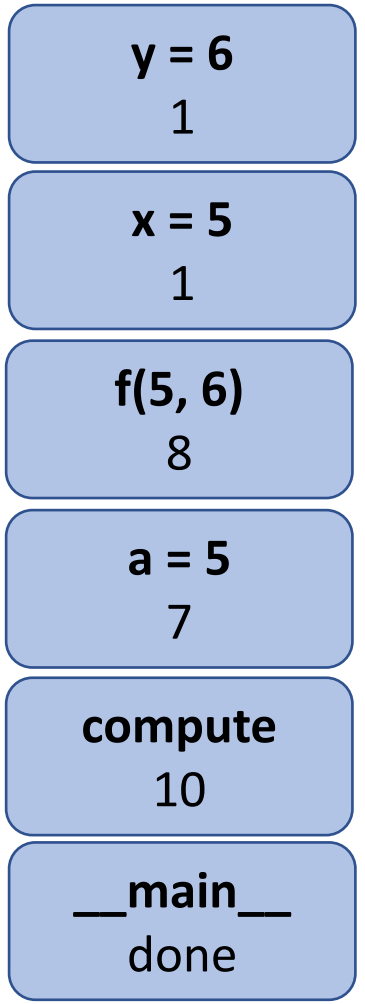
Functions and stack frames: with parameters and variables

```
1  def f(x, y):
2      print('x:', x, 'y:', y)
3      z = x + y
4      print('z:', z)
5
6  def compute():
7      a = 5
8      f(a, a + 1)
9
10 compute()
```



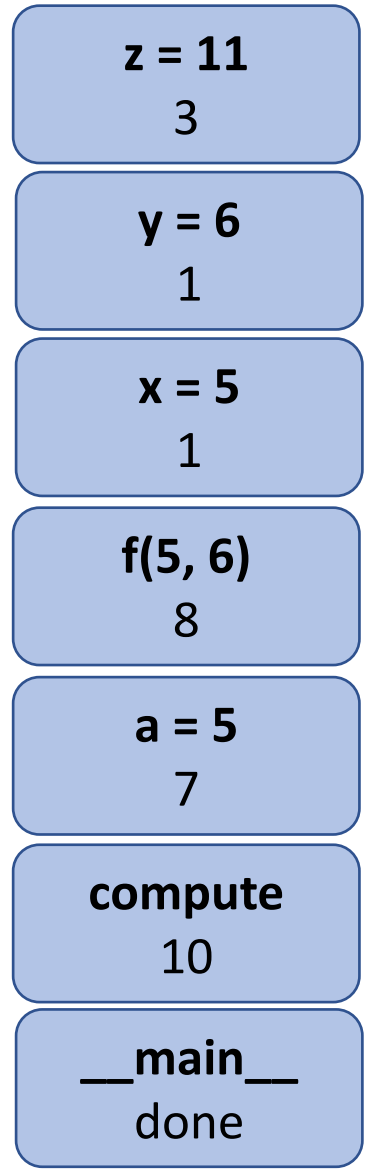
Functions and stack frames: with parameters and variables

```
1  def f(x, y):
2      print('x:', x, 'y:', y)
3      z = x + y
4      print('z:', z)
5
6  def compute():
7      a = 5
8      f(a, a + 1)
9
10 compute()
```



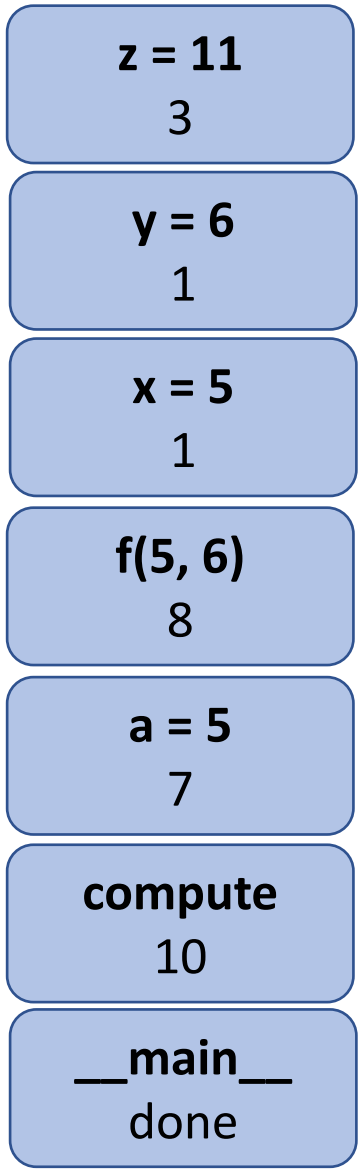
Functions and stack frames: with parameters and variables

```
1  def f(x, y):
2      print('x:', x, 'y:', y)
3      z = x + y
4      print('z:', z)
5
6  def compute():
7      a = 5
8      f(a, a + 1)
9
10 compute()
```



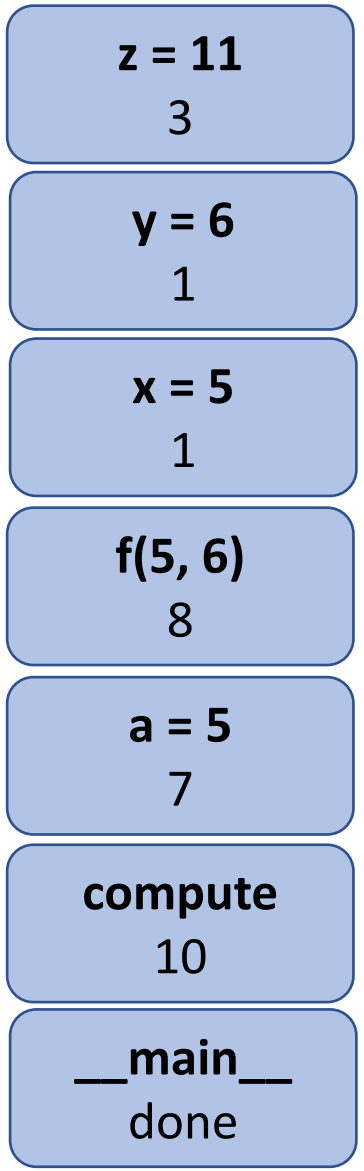
Functions and stack frames: with parameters and variables

```
1  def f(x, y):
2      print('x:', x, 'y:', y)
3      z = x + y
4      print('z:', z)
5
6  def compute():
7      a = 5
8      f(a, a + 1)
9
10 compute()
```



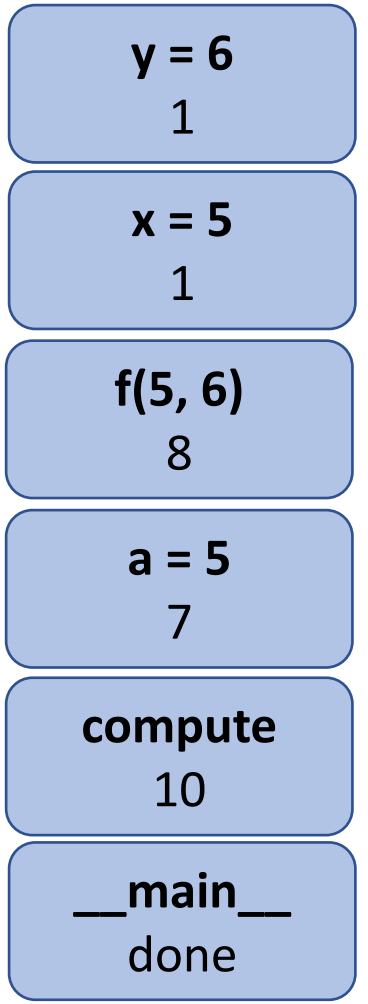
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



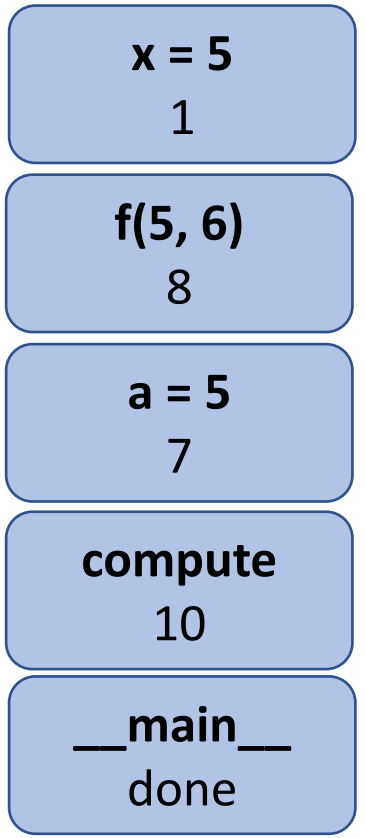
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



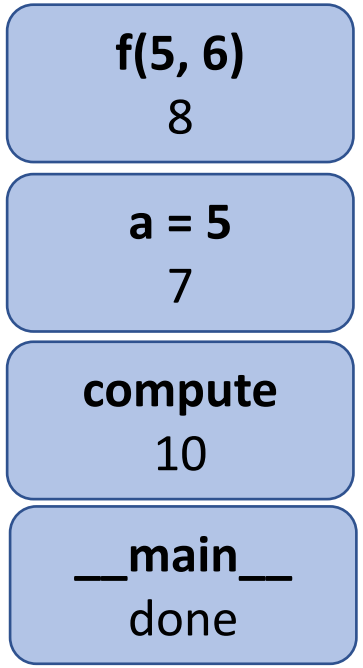
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



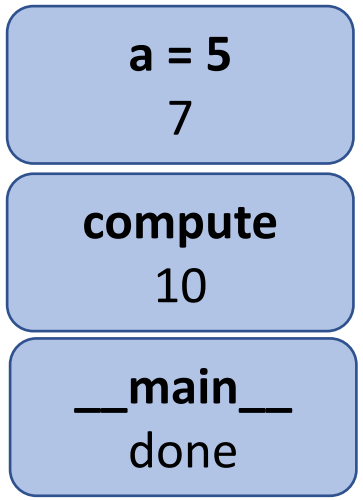
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



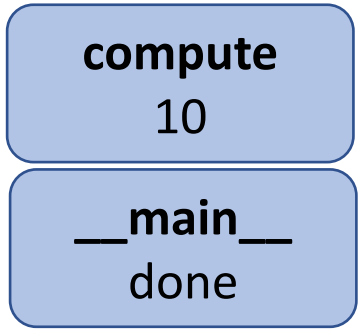
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



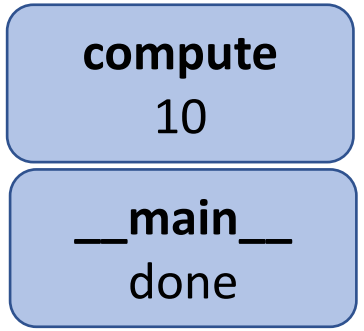
Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```



Functions and stack frames: with parameters and variables

```
1  def f(x, y):
2      print('x:', x, 'y:', y)
3      z = x + y
4      print('z:', z)
5
6  def compute():
7      a = 5
8      f(a, a + 1)
9
10 compute()
```



Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```

__main__
done

Functions and stack frames: with parameters and variables

```
 1 def f(x, y):
 2     print('x:', x, 'y:', y)
 3     z = x + y
 4     print('z:', z)
 5
 6 def compute():
 7     a = 5
 8     f(a, a + 1)
 9
10 compute()
```

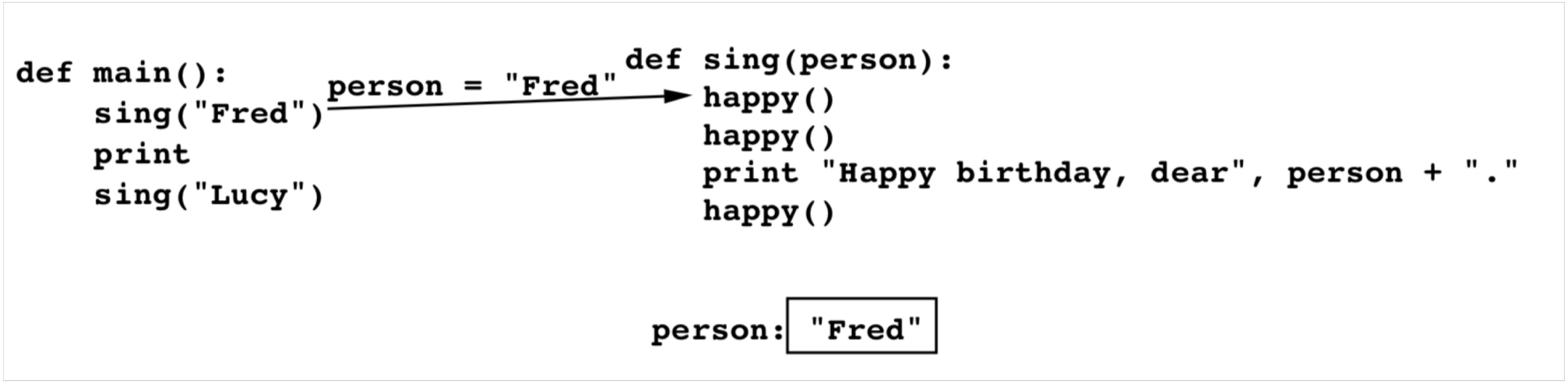
Run-time stack is empty!

Local scope of variables: happy birthday example

```
def happy():
    print "Happy birthday to you!"

def sing(person):
    happy()
    happy()
    print "Happy Birthday, dear", person + ".")
    happy()

def main():
    sing("Fred")
    print()
    sing("Lucy")

main()
```

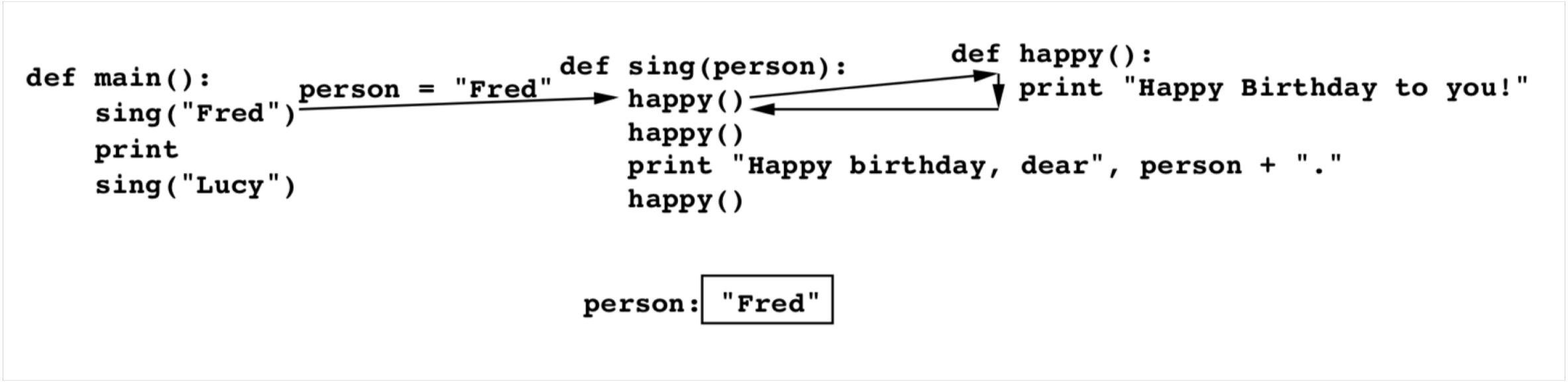
Scope of variables: happy birthday example

Control being transferred from `main()` to `sing(person)`



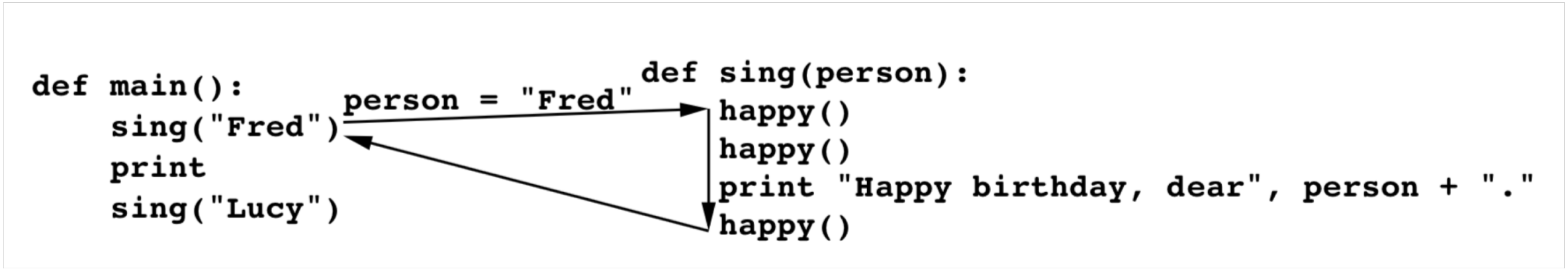
Scope of variables: happy birthday example

Execution of `sing("Fred")` starts, and the control gets further passed to `happy()`



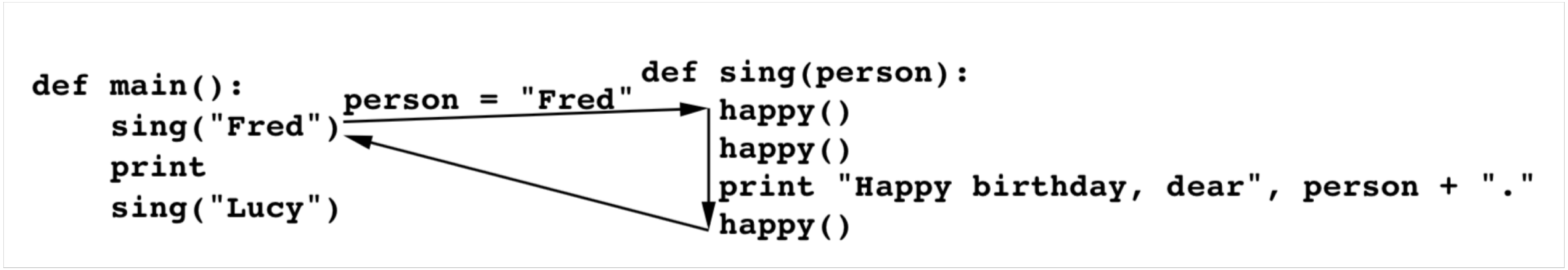
Scope of variables: happy birthday example

Call of `sing("Fred")` is completed



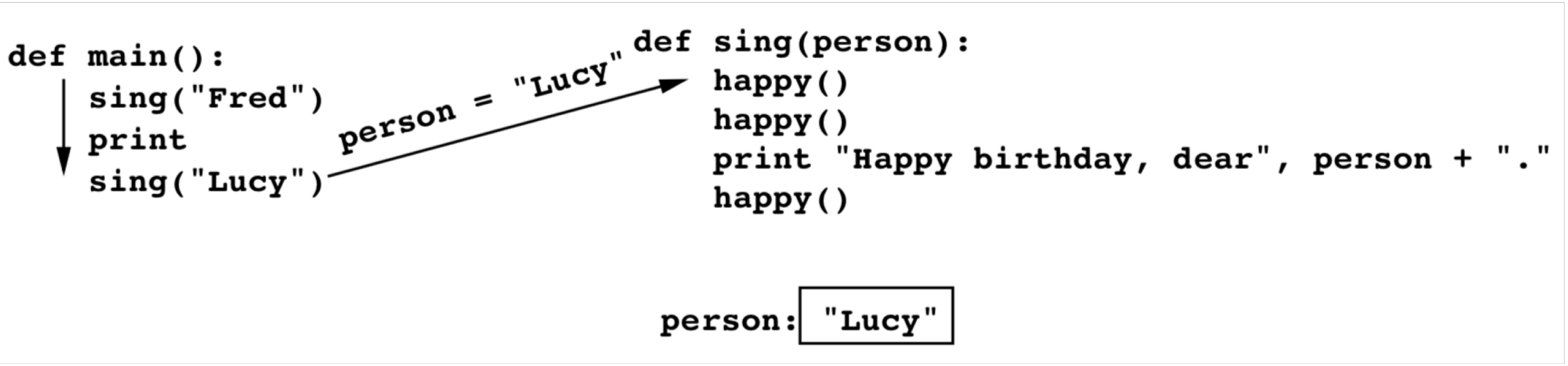
Scope of variables: happy birthday example

Call of `sing("Fred")` is completed, and the control passes back to `main()`,
the variable `person` doesn't exist anymore in the program, and it can't be referred to in `main()`



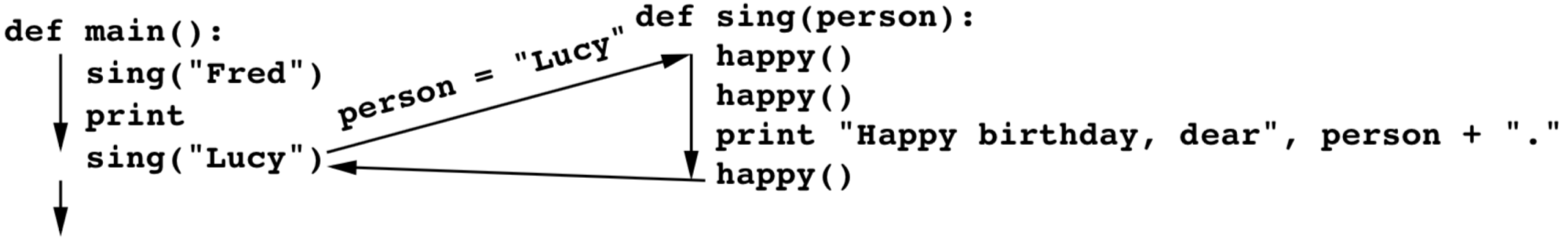
Scope of variables: happy birthday example

`main()` execution keeps going on, and eventually `sing("Lucy")` is called, in this case a new parameter variable `person` is allocated on the stack frame, with value `"Lucy"`



Scope of variables: happy birthday example

`sing("Lucy")` is executed, and finally the control is passed back to `main()`, again all run-time stack data, including are cleared up and aren't anymore accessible in the program



Moral:

All variables being created inside a function, including from the function arguments,
live the time the function is being executed,
therefore, they cannot be used outside the function

Function types: doing something and not returning values

- Functions that *do something* and do *NOT return anything* (i.e., return the default value `None`)

```
def my_print(s):
    print(s)

def add(x,y):
    print(x+y)
    return

a = my_print("Example of user-defined function that doesn't return anything")
b = add(1,2)
c = print("Print is a built-in function that displays things and returns None")
d = [8,2,6,9].sort()
e = my_list.append("New list element")
```

All variables `a,b,c,d,e` get the value `None`

Function types: doing something and returning values

- Functions that *do something* and *do return something different than* `None`

```
x = [3,2,6]
a = sorted(x)
b = min(x)
c = x.pop(1)
```

In these example cases of using built-in functions:
a is [2,3,6], b is 2, c is 2

```
def squared_sum(x,y):
    return (x*x + y*y + 2*x*y)

def sort_and_split(l):
    s = sorted(l)
    s_lower = s[:len(s)//2]
    s_upper = s[len(s)//2:]
    return [s_lower, s_upper]

a = squared_sum(2,3)
b = sort_and_split(3,2,-1,9)
```

In these example cases of user-defined functions:
a is 25 , b is [[-1,2], [3,9]]

Function types: doing something using input parameters

- Functions that *do something based on input parameters* (and returning or not something)

  All previous examples (returning and not returning something) are actually this type of functions!

```
def my_print(s):
    print(s)

def add(x,y):
    print(x+y)
    return


x = [3,2,6]
a = sorted(x)
b = min(x)
c = x.pop(1)
```

```
def squared_sum(x,y):
    return (x*x + y*y + 2*x*y)

def sort_and_split(l):
    s = sorted(l)
    s_lower = s[:len(s)//2]
    s_upper = s[len(s)//2:]
    return [s_lower, s_upper]
```

Function types: doing something not using input parameters

- Functions that *do something NOT using input parameters* (and returning or not something)

```
def hello():
    print("Hello!")

def let_some_time_passing():
    iterations = 100000000
    for t in range(iterations):
        continue
    return iterations
```

```
def beep():
    print('\a')

def get_integer():
    while True:
        string = input("Input an integer number: ")
        if string.isdigit():
            return int(string)
```

Function types: do something and modify input parameters

- Functions that do something based on input parameters and (as a side effect) modify the input parameters also in the calling (sub)program: only possible for **mutable types**, such as the `list` type!

```
15 def sort_and_get_middle(values):
16     print("Address of list variable 'values' in function: ", id(values))
17     values.sort()
18     return values[ len(values)//2 ]
19
20 x = [8, 3, 7, -1, 5]
21 print("List variable 'x' in main: ", x)
22 print("Address of list variable 'x' in main: ", id(x))
23
24 middle = sort_and_get_middle(x)
25 print("List variable 'x' in main after function call: ", x)
```

Output:

```
List variable 'x' in main:  [8, 3, 7, -1, 5]
Address of list variable 'x' in main:  4730398920
Address of list variable 'values' in function:  4730398920
List variable 'x' in main after function call:  [-1, 3, 5, 7, 8]
```

Same address!
`x` and `values` are **aliases**

`values.sort()` has changed the list variable `values` inside the function and because of the alias relation, also the variable `x` in the main program has changed accordingly

Function types: do something and modify input parameters

- How changing the value outside the function is it possible for a *mutable data type*?

➢ **Because in python arguments are always passed not by value but by reference** (the *address in memory* where their *content is stored*, see slides of Lecture 6), and in the case of *mutable data types the content at that address can "mutate", it can be updated over time*. For immutable data types any update of the variable value causes a new memory area being used for the new content, with the old one being discarded

➢ Therefore, at the call of `sort_and_get_middle(x)`, the function variable `values` (that is allocated in a new stack frame) gets the value of `x` (i.e., they point to the same memory area), that is:

the statement `values = x` is executed

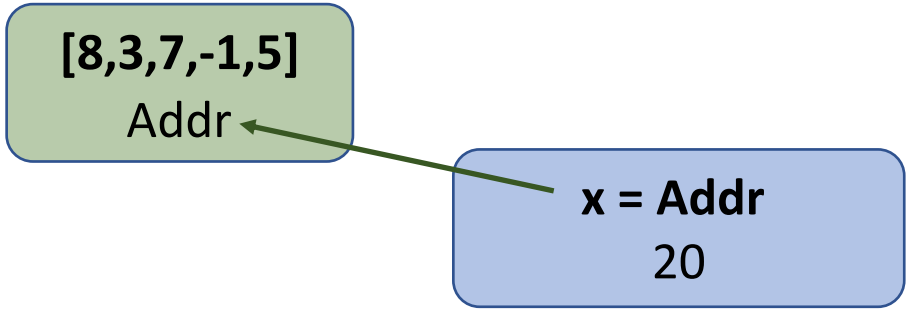
and we know that in case of mutable types, a statement such as `y = x` creates an

alias relationship between `x` and `y`

- Therefore, when the value (order) of `values` is changed in the function, this is automatically reflected also in the value (order) of `x`, outside the function, since `x` and `values` points to the same memory area
- This can be verified by noticing that `id(values)` and `id(x)` return the same (memory address) value
- When the function ends, the memory frame where the variable `values` was allocated is cleared up, but the variable `x`, with the new order, keeps existing in the program

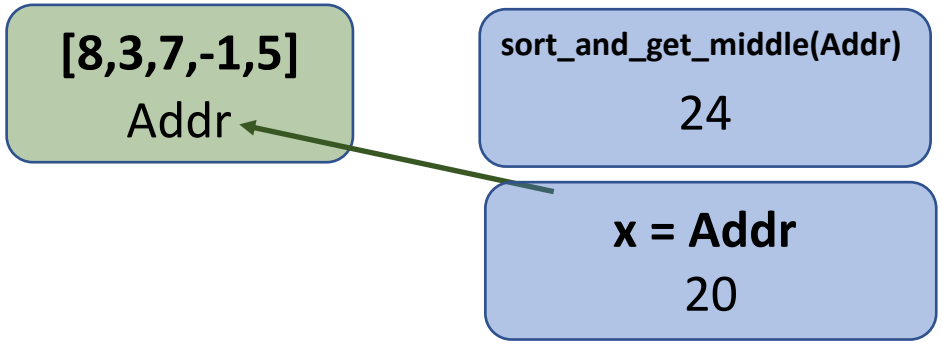
Function types: do something and modify input parameters

```
15 def sort_and_get_middle(values):
16     print("Address of list variable 'values' in function: ", id(values))
17     values.sort()
18     return values[ len(values)//2 ]
19
20 x = [8, 3, 7, -1, 5]
21 print("List variable 'x' in main: ", x)
22 print("Address of list variable 'x' in main: ", id(x))
23
24 middle = sort_and_get_middle(x)
25 print("List variable 'x' in main after function call: ", x)
```

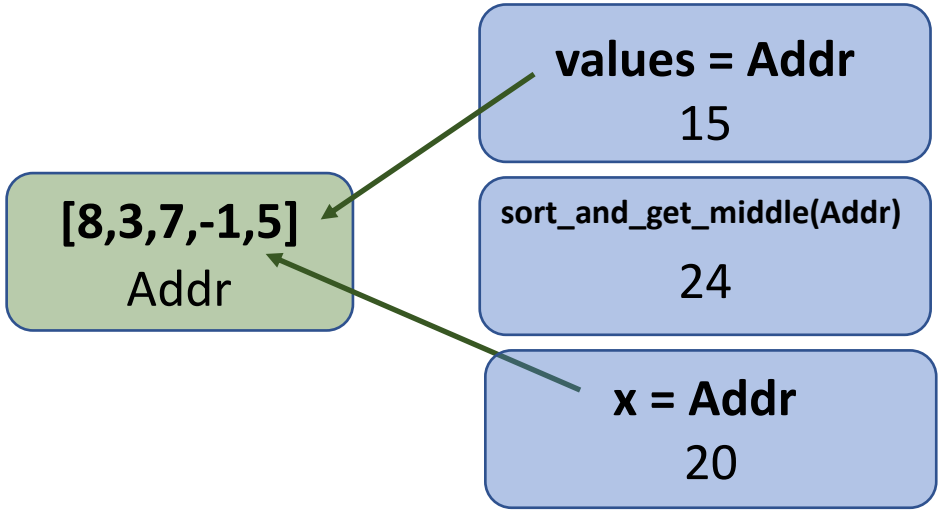
Function types: do something and modify input parameters

```
15 def sort_and_get_middle(values):
16     print("Address of list variable 'values' in function: ", id(values))
17     values.sort()
18     return values[ len(values)//2 ]
19
20 x = [8, 3, 7, -1, 5]
21 print("List variable 'x' in main: ", x)
22 print("Address of list variable 'x' in main: ", id(x))
23
24 middle = sort_and_get_middle(x)
25 print("List variable 'x' in main after function call: ", x)
```



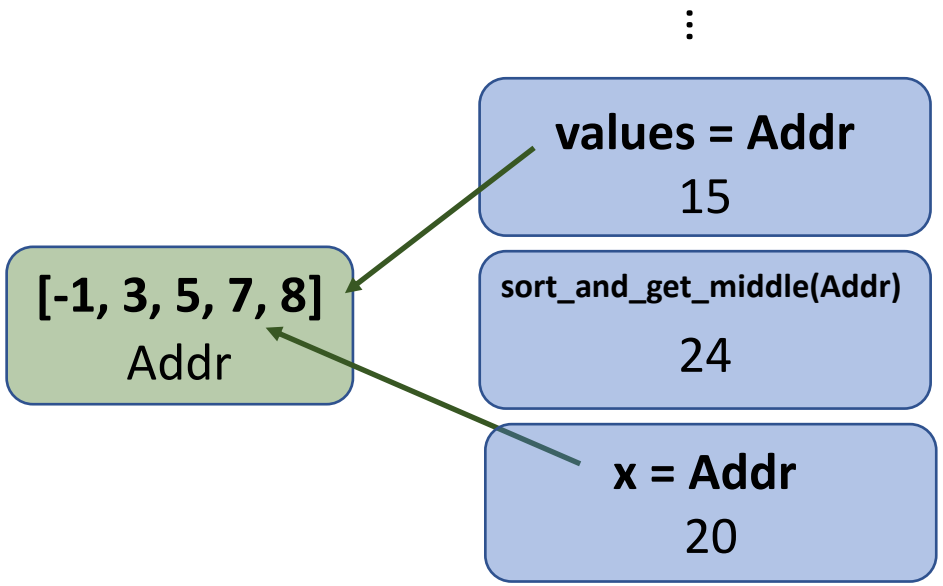
Function types: do something and modify input parameters

```
15 def sort_and_get_middle(values):
16     print("Address of list variable 'values' in function: ", id(values))
17     values.sort()
18     return values[ len(values)//2 ]
19
20 x = [8, 3, 7, -1, 5]
21 print("List variable 'x' in main: ", x)
22 print("Address of list variable 'x' in main: ", id(x))
23
24 middle = sort_and_get_middle(x)
25 print("List variable 'x' in main after function call: ", x)
```



Function types: do something and modify input parameters

```
15 def sort_and_get_middle(values):
16     print("Address of list variable 'values' in function: ", id(values))
17     values.sort()
18     return values[ len(values)//2 ]
19
20 x = [8, 3, 7, -1, 5]
21 print("List variable 'x' in main: ", x)
22 print("Address of list variable 'x' in main: ", id(x))
23
24 middle = sort_and_get_middle(x)
25 print("List variable 'x' in main after function call: ", x)
```



Function types: do something and modify input parameters

```
15 def sort_and_get_middle(values):
16     print("Address of list variable 'values' in function: ", id(values))
17     values.sort()
18     return values[ len(values)//2 ]
19
20 x = [8, 3, 7, -1, 5]
21 print("List variable 'x' in main: ", x)
22 print("Address of list variable 'x' in main: ", id(x))
23
24 middle = sort_and_get_middle(x)
25 print("List variable 'x' in main after function call: ", x)
```