15-110 Principles of Computing – F19

Lecture 17:
Functions 1

Teacher:
Gianni A. Di Caro

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)

All functions return something

```
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

Categories of functions

Returning values or not

- Functions that do something and do *not* return anything (return `None`)
- Functions that do something and do return something different than `None`

Input arguments or not / Changing arguments or not

- Functions that do something based on input parameters
  - Functions that also change the value of the input parameters (only for mutable types!)
  - Functions that do not change the value of the input parameters
- Functions that do something *not* based on input parameters

Making effects outside of the function (mutable inputs)

- Function `my_sort()` returns `None`, but it does change its input parameter `L`, which is of <u>mutable type</u>
- The function `my_sorted()` does return a NEW list, `LL` and doesn't change the input list `L`

```
def my_sort(L):
    L.sort()

def my_sorted(L):
    LL = sorted(L)
    return LL

unsorted = [1,5,0,-1, 4, -100]
unsorted2 = unsorted.copy()

my_sort(unsorted)
print(unsorted)

L = my_sorted(unsorted)
print(L)
```

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

Abstraction

Reusability

Fundamental ingredients in the design of computational solutions

Decomposition: simple interactive game example

Goal: design a simple *Guess the number* game where the user inputs three integer numbers and the system checks if she/he has guessed the right number 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 guessed it right!!!")
else:
    print("Wrong guess...")
```

Rule (hidden to the player):

- Win iff the product of the three numbers is divisible by 13

Decomposition: example

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

take the numbers in the list and make their product

check the correctness of the number 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 guessed it right!!!")
else:
    print("Wrong guess...")
```

Decomposition: example

- ✓ 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_guess(input_val):
    if not (input_val % 13):
        print(" You guessed it right!!!")
    else:
        print(" Wrong guess...")
```

```
numbers = get_user_numbers()
guess_number = make_product(numbers)
check_guess(guess_number)
```

Decomposition: example

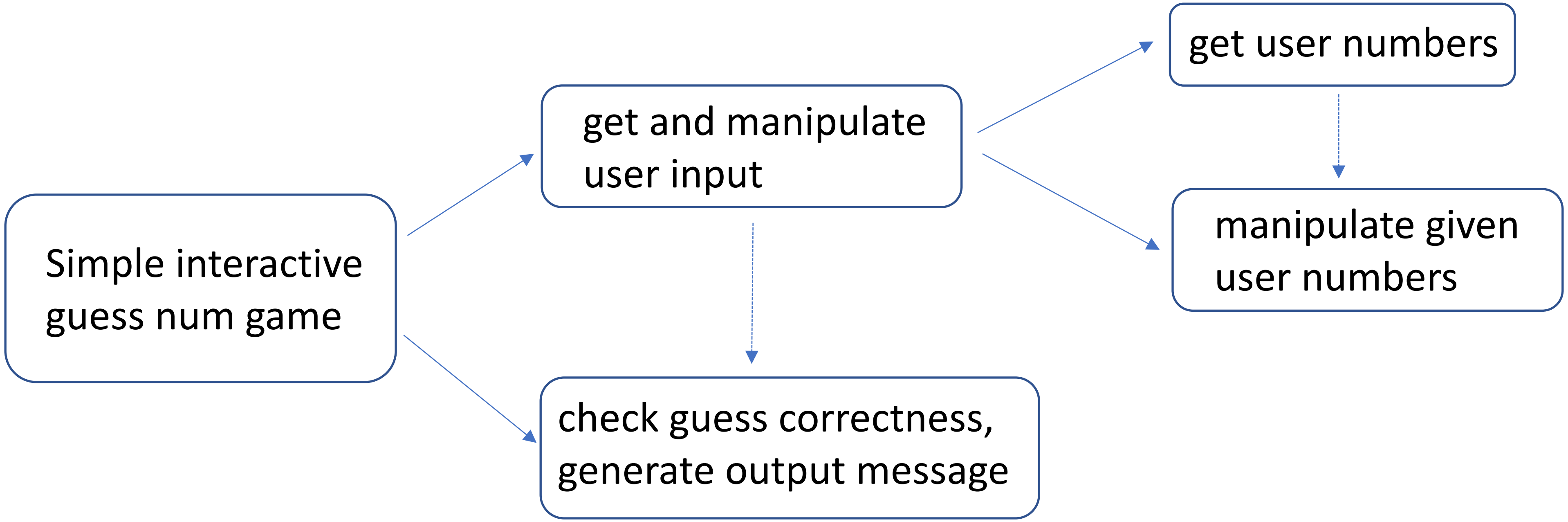
```
def get_user_numbers():
    value_list = []
    n = 0
    while n < 3:
        input_str = input('Input an integer number: ')
        if input_str.isdigit():
            v = int(input_str)
            value_list.append(v)
            n += 1
    return value_list

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

def check_guess (input_val):
    if not (input_val % 13):
        print("You guessed it right!!!")
    else:
        print("Wrong guess...")
```

```
numbers = get_user_numbers()
guess_number = make_product(numbers)
check_guess(guess_number)
```

- Even better:

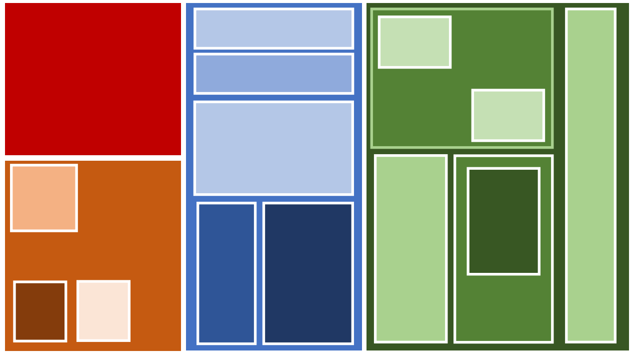
```
def get_number():
    values = get_user_numbers()
    number = make_product(values)
    return number

guess_number = get_guess_number()
check_guess(guess_number)
```

Decomposition using functions



Problem to solve
(monolithic view)

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

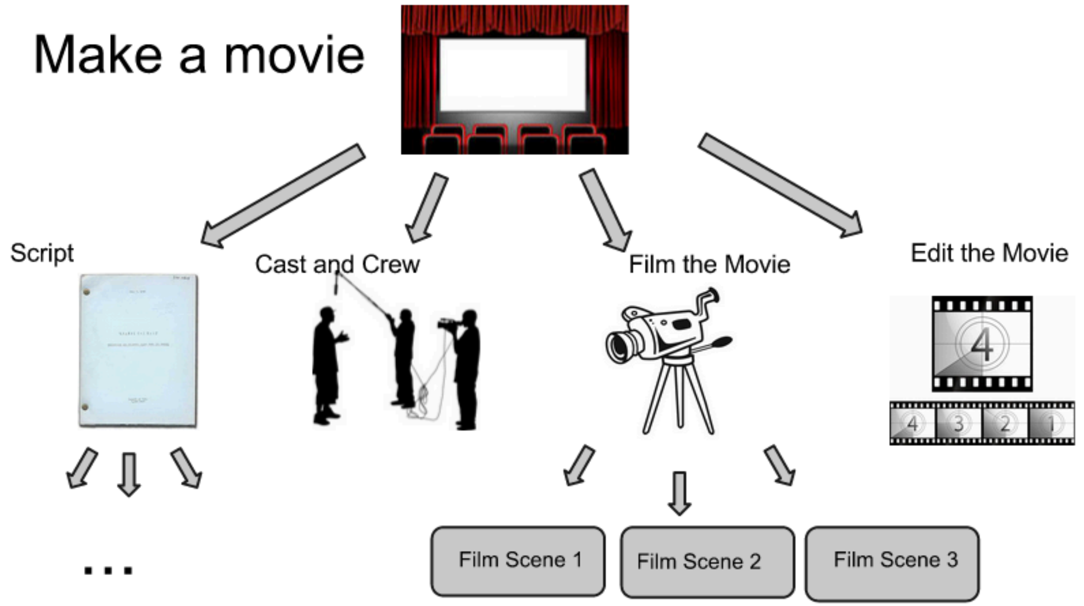


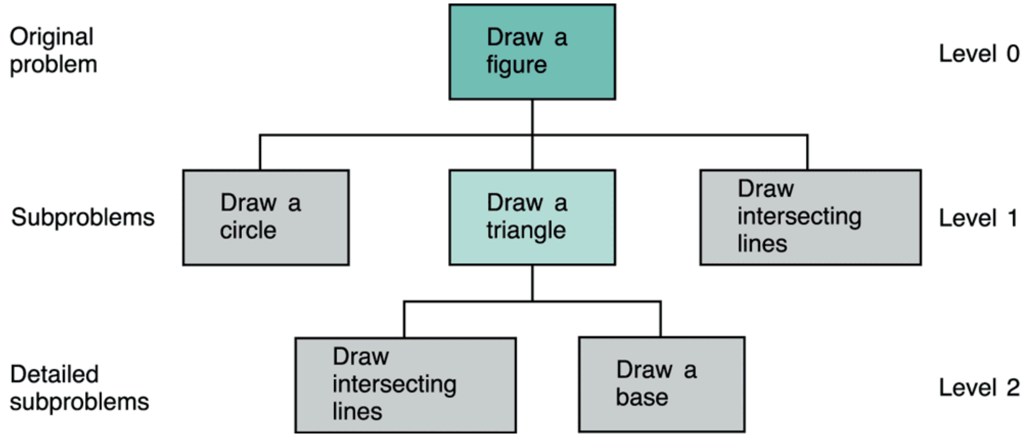
Divide-and-conquer

Gains from decomposition using functions

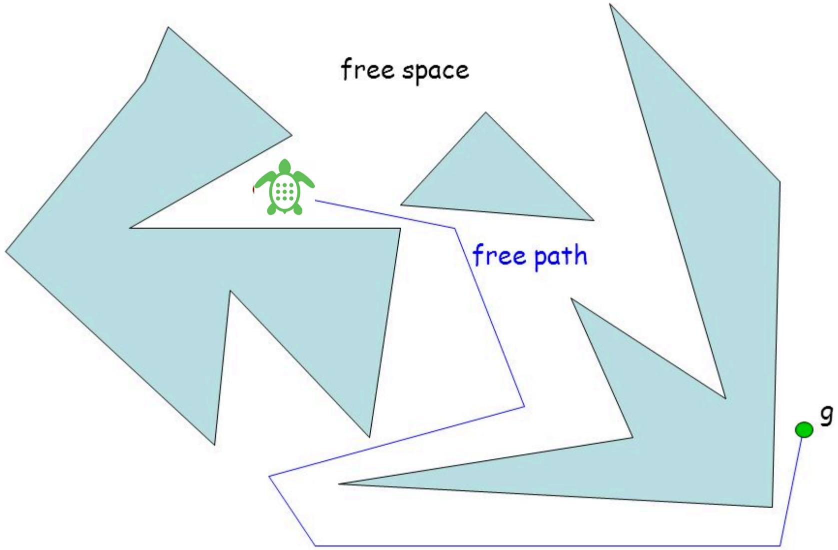
- Gains resulting from decomposing the problem in functional blocks and defining functions / primitives to handle the computation in each block:
  - ✓ A better **understanding of the task** and of **our computational approach** to its solution
  - ✓ **Readability**: we can now *read* what's going on (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





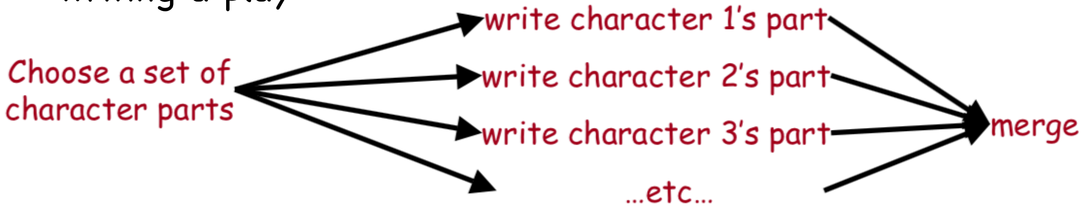
designing a restaurant menu



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!

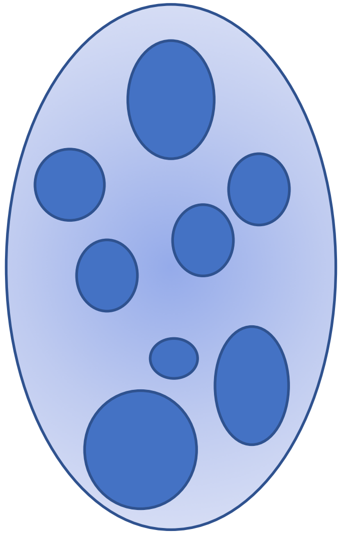
writing a play



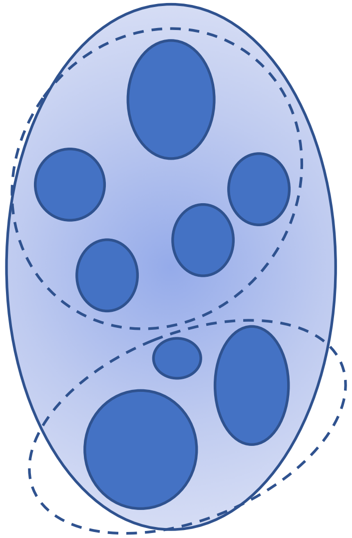
Decomposition by creation of self-contained modules

- Decomposition can be realized at <u>different granularity</u>, depending on the task domain

Identification of *functional blocks*, definition of new **primitive functions**



Organization of defined functions in *self-contained* **modules**



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()
```

- ✓ Defines an *abstraction*: it describes a computation that applies to <u>any</u> 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?

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()
```

```
sing("Fred")
print()
sing("Lucy")
```

```
sing("Fred")
print("The song was for ", person)
sing("Lucy")
```

Ok!

Local Scoping: where do function variables live?

Examples from the class notebook: read them, understand them, run them, play with them ...

```
def avg_two(x, y):
    x += 1
    y += 1
    print(x,y)
    avg = (x + y) / 2
    return avg

x = 2
y = 5
s = avg_two(x, y)
print(s, x, y)
```

```
def avg(n_list):
    val = sum(n_list) / len(n_list)
    #print("Avg is:", val)
    return val

def happy():
    a = print("Happy birthday!")
    print(a)

x = [1,2,3,4,5]
y = avg(x)
print(y)
a = 5
y = (y + a)/2
print(y)
happy()
```

```
def avg(n_list):
    for i in range(len(n_list)):
        n_list[i] += 1
    val = sum(n_list) / len(n_list)
    return val

x = [1,2,3,4]
res = avg(x)
print(res)
print(x)
```

```
def avg_two(x, y):
    x += 1
    y += 1
    print(x,y)
    avg = (x + y) / 2
    return avg

def prod_three(a,b,c):
    return a * b * c

def no_sense():
    a = 2 + hhh
    b = 3
    return a * b

hh = 6
print(no_sense())
hhh = 3
```

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**

… complicated (maybe) … let’s skip the technical details

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)`

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

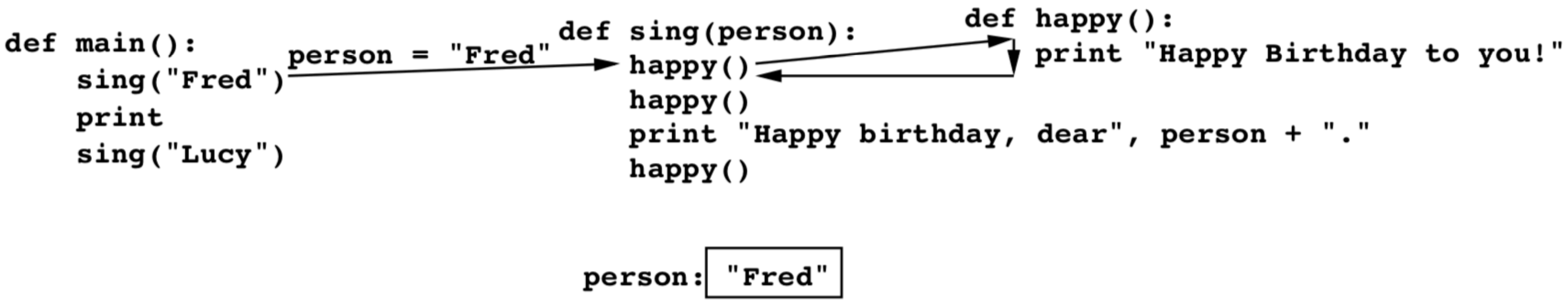
person = "Fred"

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

person: "Fred"

Scope of variables: happy birthday example

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



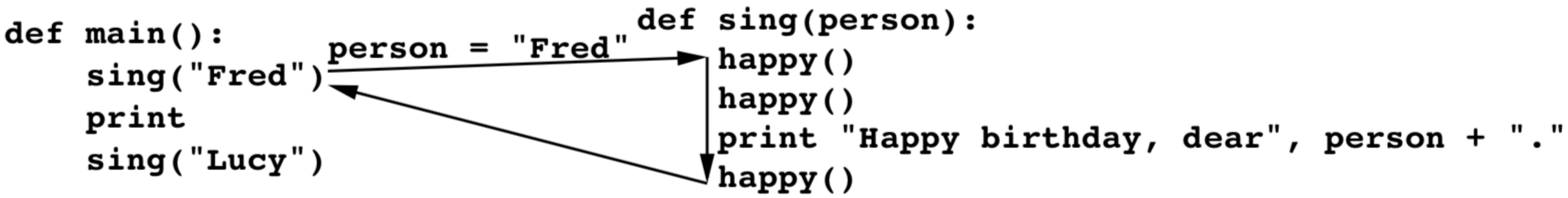
Scope of variables: happy birthday example

Call of `sing("Fred")` is completed, and the control passes back to `main()`

```
def main():                 person = "Fred"   def sing(person):
    sing("Fred")                                  happy()
    print                                         happy()
    sing("Lucy")                                  print "Happy birthday, dear", person + "."
                                                  happy()
```

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

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

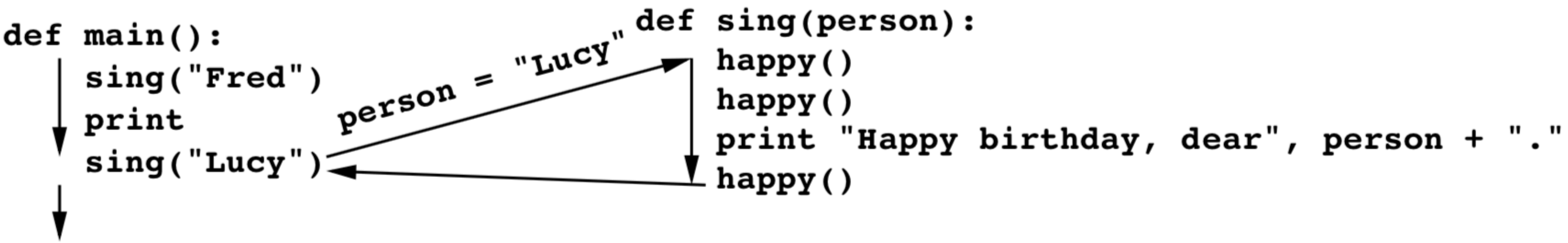
person = "Lucy"

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

person: $\boxed{\texttt{"Lucy"}}$

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 are cleared up and aren't anymore accessible in the program



Moral:

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

How do we pass arguments to a function?

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

- If we only need to pass one **single argument**, there's no much to say
- The situation is different when passing **multiple parameters** …

```
def quadratic_roots(a, b, c):
    x1 = -b / (2 * a)
    x2 = sqrt(b**2 - (4 * a *c)) / (2 * a)
    return (x1 + x2), (x1 - x2)
```

✓ Passing arguments as **positional arguments**

```
quadratic_roots(31, 93, 62)
```

is different than:

```
quadratic_roots(93, 31, 62)
```

Order matters!

✓ Passing arguments as **keyword arguments**

```
quadratic_roots(a=31, b=93, c=62)
```

is the same as:

```
quadratic_roots(b=93, a=31, c=62)
```

Order doesn't matter!

Keyword arguments and the `help()` function

- Passing arguments as **keyword arguments** works because python **knows the name function arguments**, and therefore it can perform automatic matching without errors

```
quadratic_roots(a=31, b=93, c=62)
quadratic_roots(b=93, a=31, c=62)
quadratic_roots(c=62, b=93, a=31)
```

all give the same result, (-1.0, -2.0) in this case

→ We can ask python **help** on function's parameters using the `help(function_name)` function:

`help(quadratic_roots)` would give as answer: `quadratic_roots(a,b,c)`

Use of keyword arguments increases the clarity of a program!

`random_password(upper=1, lower=1, digits=1, length=8)` vs. `random_password(1, 1, 1, 8)`

Positional arguments: different possible errors

- When passing arguments as **positional arguments** we need to be careful to **match the order** with which the parameters appear in the function definition!

  - Wrong computations (no errors are issued by the interpreter!)

```
quadratic_roots(31, 93, 62) → (-1.0, -2.0)

quadratic_roots(62, 93, 31) → (-0.5, -1.0)
```

  - Run-time errors due to incorrect type (program aborted!)

```
def sing(person, repetitions):
    for i in range(repetitions):
        happy()
        happy()
        print("Happy Birthday, dear", person + ".")
        happy()
```

```
sing(2, “Fred”)
```

Throws an error because a string object cannot be interpreted as an integer

Default values for the arguments, equivalent function calls

- When defining a function, a **default value can be defined for each argument**
  - If a value argument for an argument with a default value is passed (either by position or by keyword) when the function is called, then the argument takes the provide value
  - Otherwise, the argument takes the default value

- `def sing(person="Fred", repetitions=2):`
  - ✓ `sing()`
  - ✓ `sing(“Lucy”)`
  - ✓ `sing(“Lucy”, 3)`
  - → `NO: sing(3)`

- `def sing(person, repetitions=2):`
  - → `NO: sing()`
  - ✓ `sing(“Lucy”)`
  - ✓ `sing(“Lucy”, 3)`
  - → `NO: sing(3)`

→ `NO: def sing(person="Fred”, repetitions):` parameter assignments would be ambiguous

Default values for making arguments optional

- The parameters with default values are de facto **optional**, in the absence of them they take the default value, that might be an empty value

  - `def sing(person, repetitions = 2):`
  - `def draw_rectangle(x1, x2, y1, y2, fill_color = None):`
  - `def move_forward(distance, velocity = 10):`

Arbitrary number of arguments

- It might be the case that a function does some repetitive job and operates on a non well-defined number of arguments
- E.g., `print()` function
- We could use lists but it’s not always nice, convenient, appropriate pack everything into a string
- Arbitrary sequence of arguments can be passed with the notation `*arguments`

```
def longlen(*strings):
   max = 0
   for s in strings:
     if len(s) > max:
        max = len(s)
   return max

longlen('apple','banana','cantaloupe','cherry’)  → 9
longlen(‘red’,’blue’,’green’)                    → 5
```

- Positional and keyword arguments should be placed first to the arbitrary ones

```
def my_func(a, b=True, *args):
```

Passing functions as arguments!

- Function arguments can also include a **function, that can be then regularly invoked inside the function**

```
def parabola(x):
    return -x*x + 2

def cubic(x):
    return x*x*x + 2*x*x

def geometric(x):
    return 1 / (1-x)

def line(x):
    return x
```

```
def estimate_max_in_interval(f, low_val, high_val, samples):
    x = low_val
    step = (high_val - low_val) / samples
    max_val = f(high_val)
    for i in range(samples):
        if f(x) > max_val:
            max_val = f(x)
        x += step
    return max_val

print(estimate_max_in_interval(geometric, -2, -1, 100))
```

Abstraction

- **Abstraction** in computing aims to **hide details that are not necessary in a given context**, preserving only the information that is relevant in the context: it is the process that allows to take a piece of code, *name it*, and use it as it were a *black-box*

- When we define a function we are performing an **abstraction**: we take a piece of code, including objects and expressions, we name it, and in principle we can we use it, without caring about *how* the outputs are precisely obtained in the function body

- The only information relevant to use the function are its input parameters and the returning object types, the details about how processing is performed are hidden by the abstraction

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

```
def make_product(values):
    p = 1
    for i in range(len(values)):
        p *= values[i]
    return p
```

```
import numpy
def make_product(values):
    return numpy.prod(values)
```

❖ All these three functions do the same thing! All that matters for the user are the **type of the inputs**, `values`, and what the function **returns**. *How* things are done inside the function doesn't matter for using it!