





Programming 1

Introduction in
programming
Lecture 6

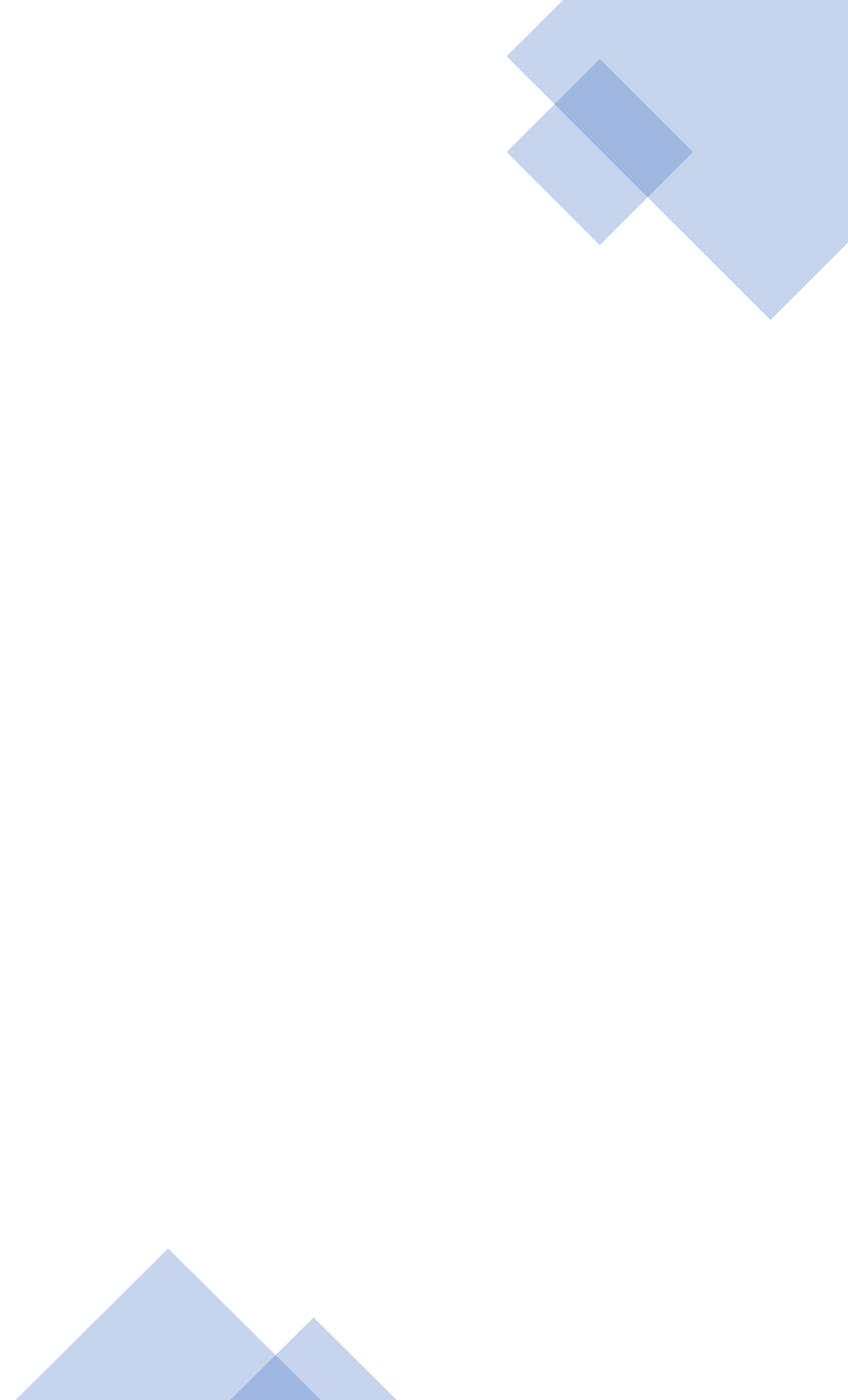


What we talked about?

- Loops
 - Data Structures
 - Strings
 - Regular expression module
- 
- 



What will we talk about?

- Functions
 - Function definition
 - Functions call
 - Variables
 - Local variables
 - Global variables
- 

Let us consider the following example

A Flashlight



Decomposition

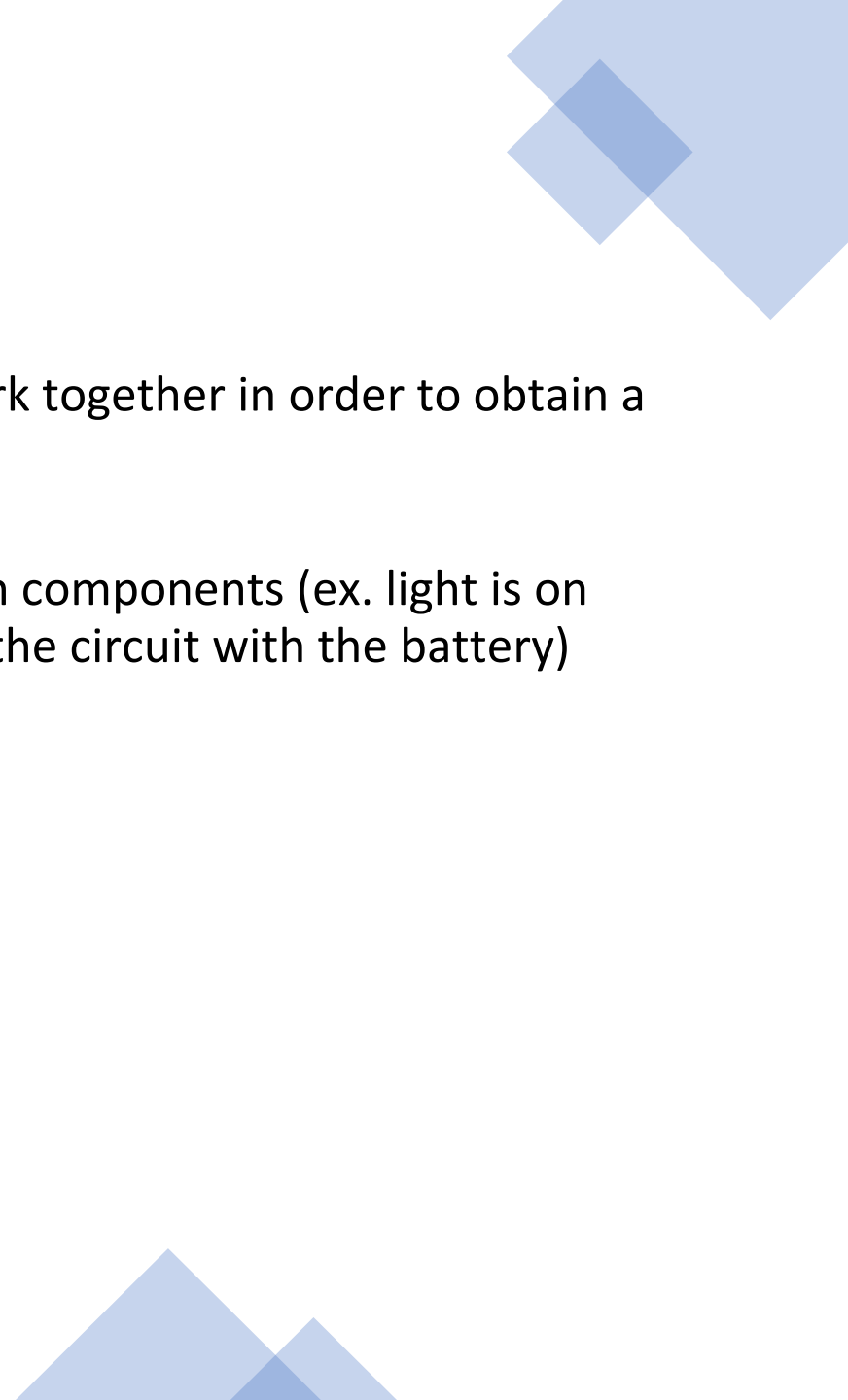
Two perspectives

- How is it functioning?
 - What components does it have?
 - How do these components interact?
- How can it be used?
 - What to know in order to use it?
 - How to start/stop?

Abstractisation

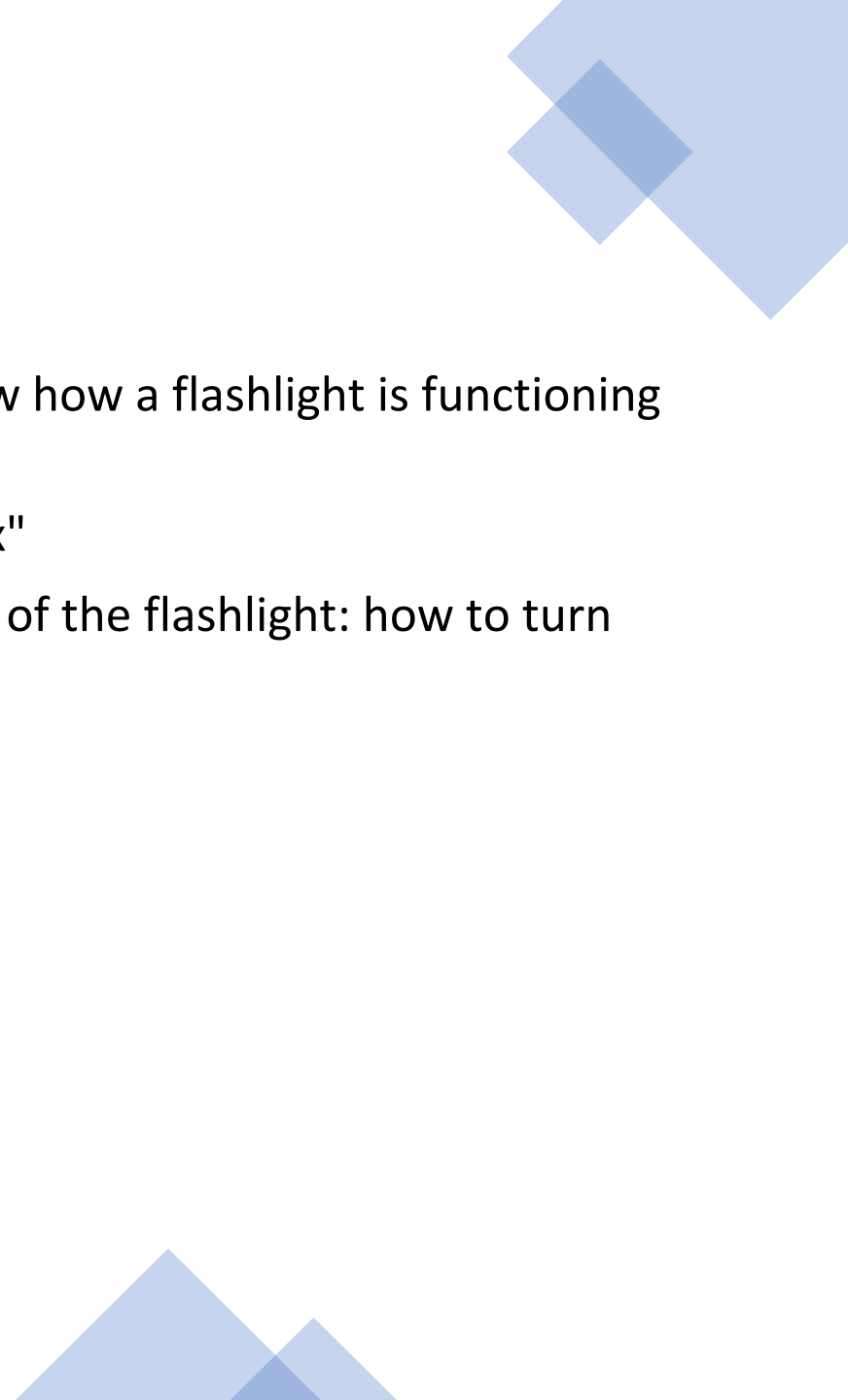


Decomposition

- Multiple components work together in order to obtain a result
 - Clear interaction between components (ex. light is on after the switcher closes the circuit with the battery)
- 




Abstractisation

- It is not necessary to know how a flashlight is functioning in order to use it
 - A flashlight is a "black box"
 - We know the "interface" of the flashlight: how to turn on/off
- 


Abstractisation

<https://www.youtube.com/watch?v=K6NgMNvK52A>



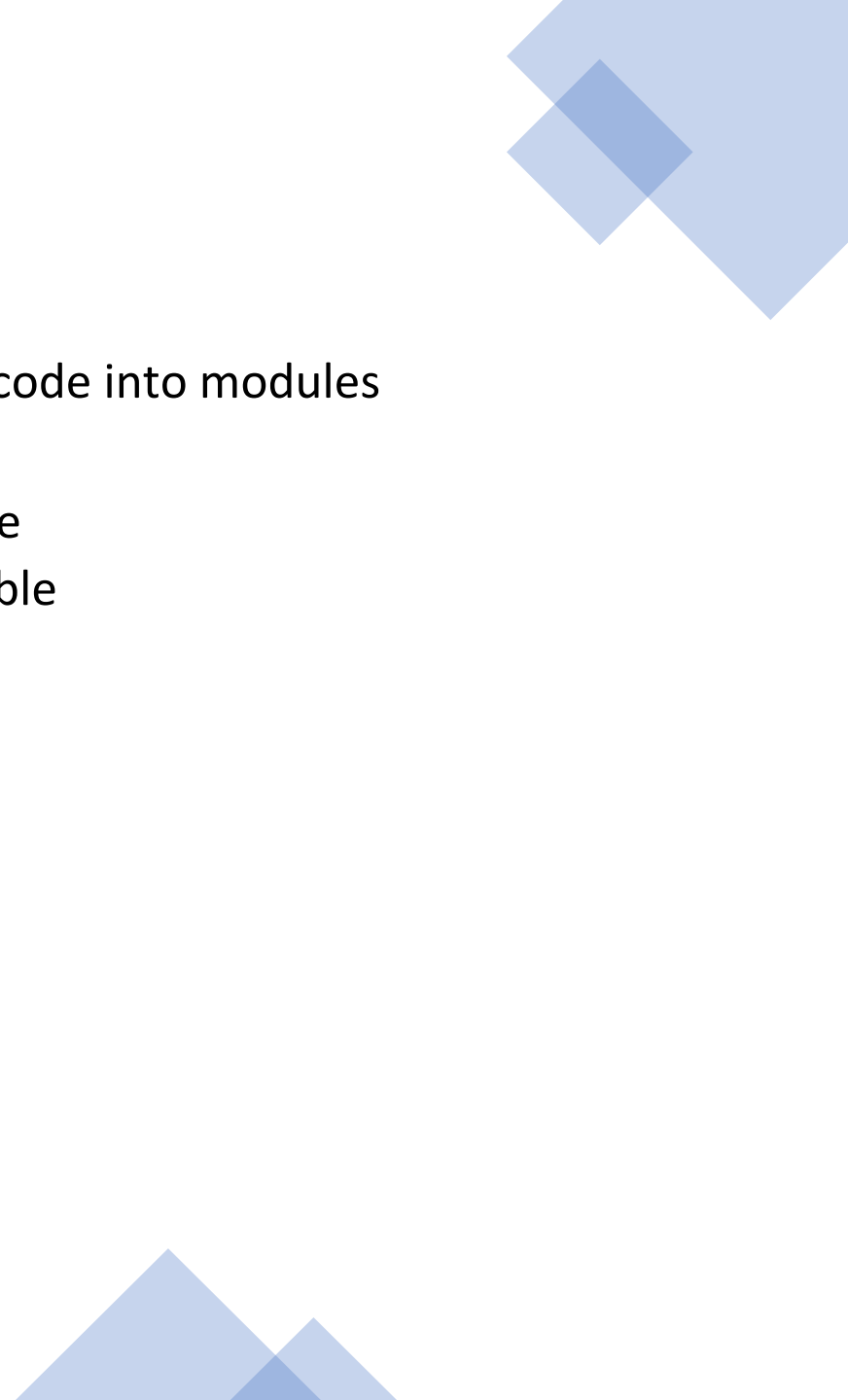


Apply these principles
when you are
programming!



Add STRUCTURE Using DECOMPOSITION

In programming, we divide code into modules

- are self-contained
 - used to break up code
 - intended to be reusable
 - keep code organized
 - keep code coherent
- 




Hide DETAILS Using ABSTRACTIZATION

Flashlight:

- minimal user manual in order to use
- it is not necessary to present the scheme of circuitry

In programming, think of a piece of code as a black box

- cannot see details
 - do not need to see details
 - do not want to see details
 - hide complex coding details
- 

Functions

Piece of reusable code

Functions are not executed by a program unless they are called (invoked)

- A function has the following characteristics
 - has a name
 - has parameters (0 or more)
 - has a docstring (documentation string: optional but recommended)
 - has a body
 - returns something

How to write and call a function?

```
def is_even (i):  
    """  
    Input: i, a `int` value  
    Return True if i is even or False if it is odd  
    """  
    print ("In is_even function")  
    return i%2 == 0
```

```
is_even (3)
```

Later, call the function and
assign values to arguments

Key word

Name

Parameters
Or
Arguments

Docstring

Function
Body

Function Body

```
def is_even (i):  
    """  
    Input: i, a `int` value  
    Return True if i is even or False if it is odd  
    """
```

```
    print (" In is_even function ")
```

```
    return i%2 == 0
```

Key word

Expression that is
evaluated
And
Return type

Statement




Specifications/Docstring

Are a contract between implementer of the function and user

Assumptions

- conditions that must be met by users of function. Typically constraints on parameters, such as type, and sometimes acceptable ranges of values

Guarantees

- Conditions that must be met by function, provided that it has been called in way that satisfies assumptions
- 

Variable – Visibility domain

- Formal parameter gets bound to the value of actual parameter when function is called

```
def f(x):  
    return x + 2
```

*Formal
Parameter*

```
n = 3  
z = f(n)
```

*Actual
Parameter*

Variable – Visibility domain

- Formal parameter gets bound to the **value** of actual parameter when function is called

```
def f(x):  
    Formal  
Parameter  
    x = x + 5  
    return x + 2
```

```
x = 3  
z = f(x)  
    Actual  
Parameter
```


Variable – Visibility domain

- New scope/frame/environment created when enter a function
- Scope is a mapping of names to objects in memory

```
def f(x):  
    x = x + 1  
    print ("in f(x): x =", x)  
    return x
```

Formal
Parameter

```
x = 3
```

```
z = f(x)
```

Actual
Parameter

Function
definition

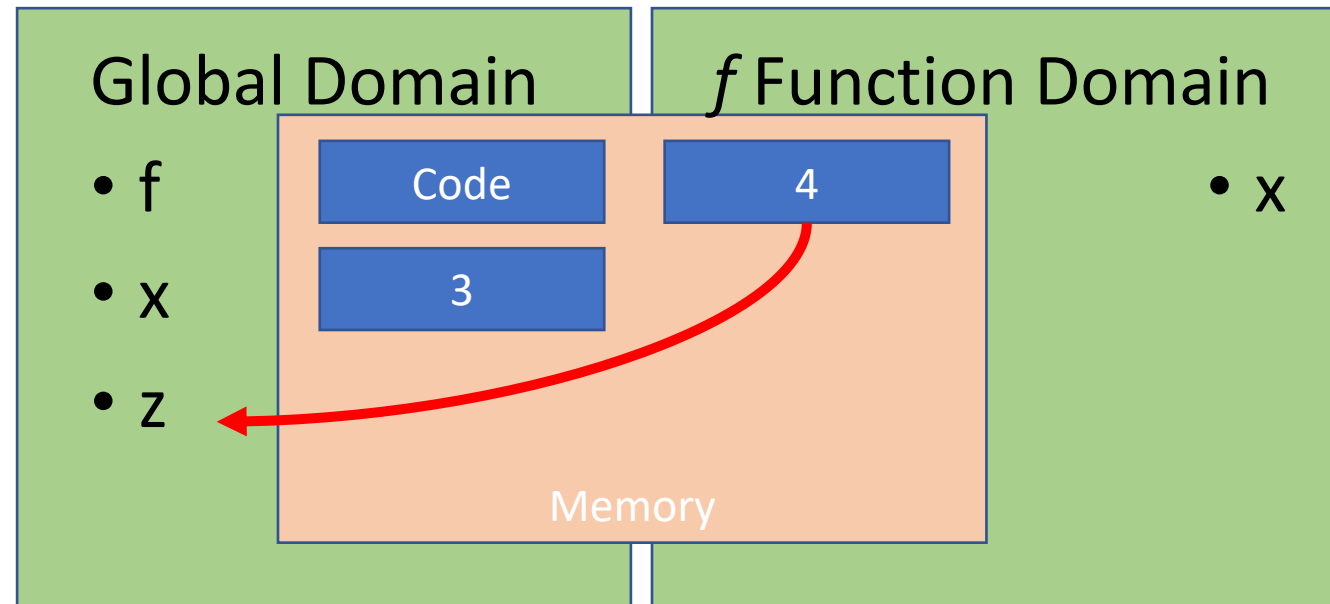
Main Program:


- Initialize variable x
- Call function f
- Assign the result returned by the function to variable z

Variable – Visibility Domain

```
def f(x):  
    x = x + 1  
    print ("in f(x): x =", x)  
    return x
```

```
x = 3  
z = f(x)
```





What happens if the function does not return any value?

Python returns the value *None*, if no return given

Represents the absence of a value

None is a special constant in the language

None is used like *NULL*, *void*, or *nil* in other languages



Return vs print

return

- only has meaning inside a function
- function terminates with return
- has a value associated with it, given to function caller

print

- can be used outside functions
- Can execute many print statements without termination
- has a value associated with it, output to the console

Functions as Function Arguments

Arguments can have any type, including functions

```
def a():  
    print ("In function a")  
def b(y):  
    print ("In function b")  
    return y  
def c(z):  
    print ("In function c")  
    return z()  
  
print (a())  
print (5+b(2))  
print (c(a))
```



[Optional] Default Values for arguments

Remember *range()* function?

How can we use it?

- `range(10)`
- `range(1,10)`
- `range(1, 10, 2)`

How this function is defined?



[Optional] Default Values for arguments

Default values for a function's arguments

These arguments are optional when the function is called

```
def my_dummy_range(start, stop, step=1):  
    l=[]  
    el = start  
    while el < stop:  
        l.append(el)  
        el = el + step  
    return l
```

```
print(my_dummy_range(1,10))  
print(my_dummy_range(1,10,1))
```

[Optional] Default Values for arguments

Default arguments must be last in the parameter list

```
def my_dummy_range(start=0, stop, step=1):  
    while el < stop:  
        ...
```



```
def my_dummy_range(start, stop=0, step=1):  
    while el < stop:  
        ...
```

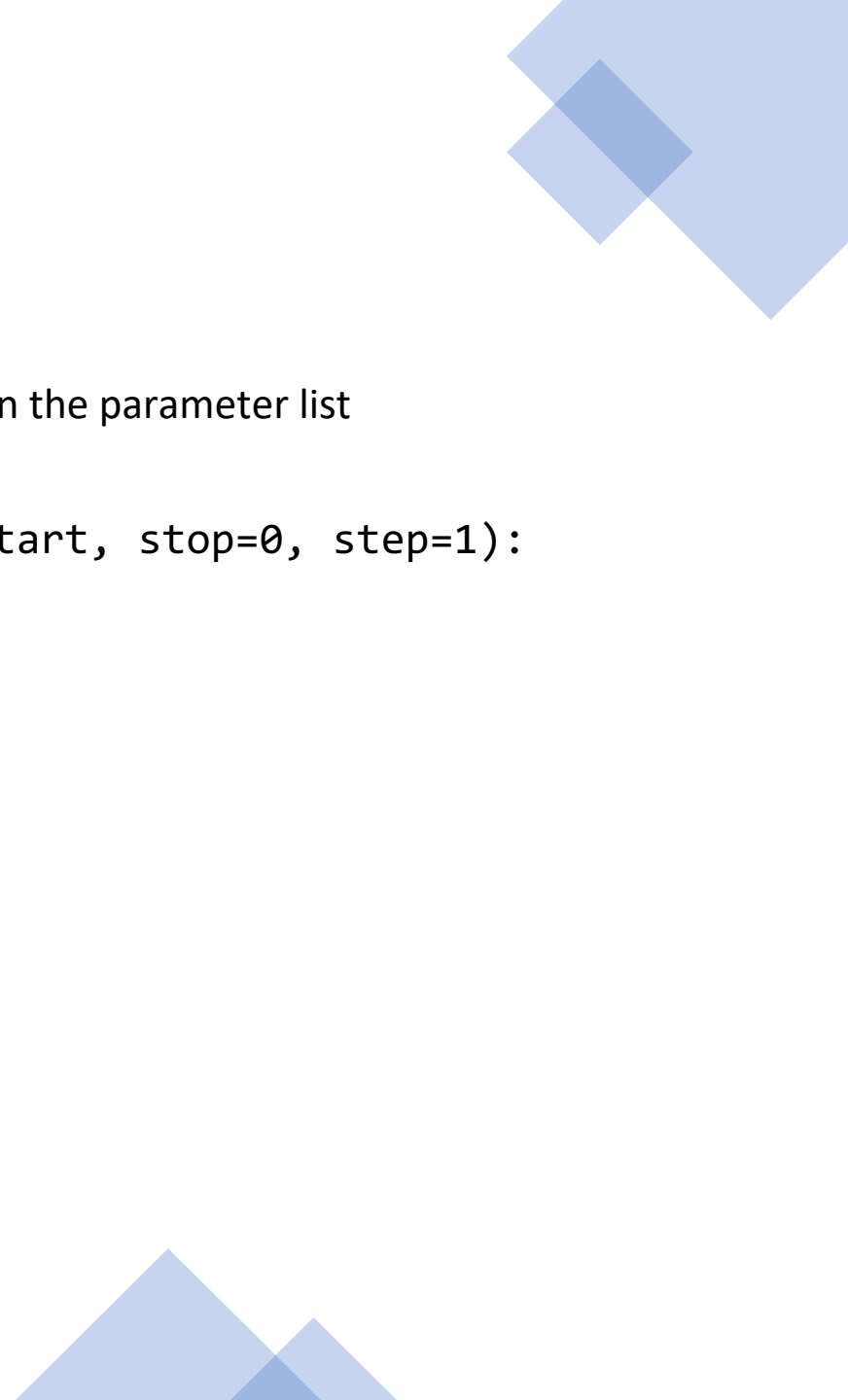




[Optional] Default Values for arguments

Default arguments must be last in the parameter list

```
def my_dummy_range(start, stop=0, step=1):  
    # let's code  
    while el < stop:  
        ...
```



[Optional] Lambda notations

Python's lambda creates anonymous functions

```
f = lambda z: z * 42  
f(7)
```

```
g = (lambda x,y: x+y)(2,3)  
print(g)
```

Only one expression in the lambda body; its value is always returned

Python supports functional programming idioms: map, filter, closures, continuations, etc.

[Optional] Lambda notation – map, reduce, filter

OUTPUT

[2, 3, 4, 5]

[101, 202, 303, 404]

[1, 3]

10

```
def add1(x): return x+1
def odd(x): return x%2 == 1
def add(x,y): return x + y

print(list(map(add1, [1,2,3,4])))
print(list(map(add, [1,2,3,4], [100,200,300,400])))
print(list(filter(odd, [1,2,3,4])))
import functools
functools.reduce(add, [1,2,3,4])
```



Visibility domain

All computer languages have scope rules

Scope rules specify which variables can be seen

Python: These rules can be summarized as LEGB:

1. Local
2. Enclosing
3. Global
4. Built-in

The search order matters: first search Local, then Enclosing, Global, and Built-in





Local Scope

Always search Local Scope first

Local Scope refers to names assigned in any way within a function, that are not declared as global






Global Scope

Global scope is searched after Local[, and Enclosed]

Global scope is simplest to understand

A name declared at Global scope, is not enclosed in a function

```
x = 100  
print(x)
```



Declaring Variables / Variables Scope

A variable name must be defined before it is used

```
print(x)  
x = 100
```

```
NameError                                Traceback (most recent call last)  
<ipython-input-15-5065468fbb19> in <module>()  
----> 1 print(x)  
      2 x = 100
```

NameError: name 'x' is not defined

Local Again

- Local Scope: names assigned in a function
- Local is searched first

```
x = 99
y = 17
def fun(x):
    y=100
    print (x, y)
fun(77)
print (x, y)
```

Result?
77 100
99 17

Local Again

- Local Scope: names assigned in a function
- Local is searched first

```
x = 99
y = 17
def fun(x):
    y=100
    print (x, y)
fun(77)
print (x, y)
```

x Variable values

```
99 <- global declaration
77 <- function call
(function parameter
hides global x name)
99 <- global declaration
```

y Variable values

```
17 <- global declaration
100 <- function body
declaration
17 <- global declaration
```

Result?

77 100

99 17

Keyword global

- A global variable can be declared in a function using the keyword global
- Caution: function has to be called in order to define the variable, otherwise the variable is never defined

```
def fun():  
    global x  
    x=100
```

```
fun()  
print (x)
```