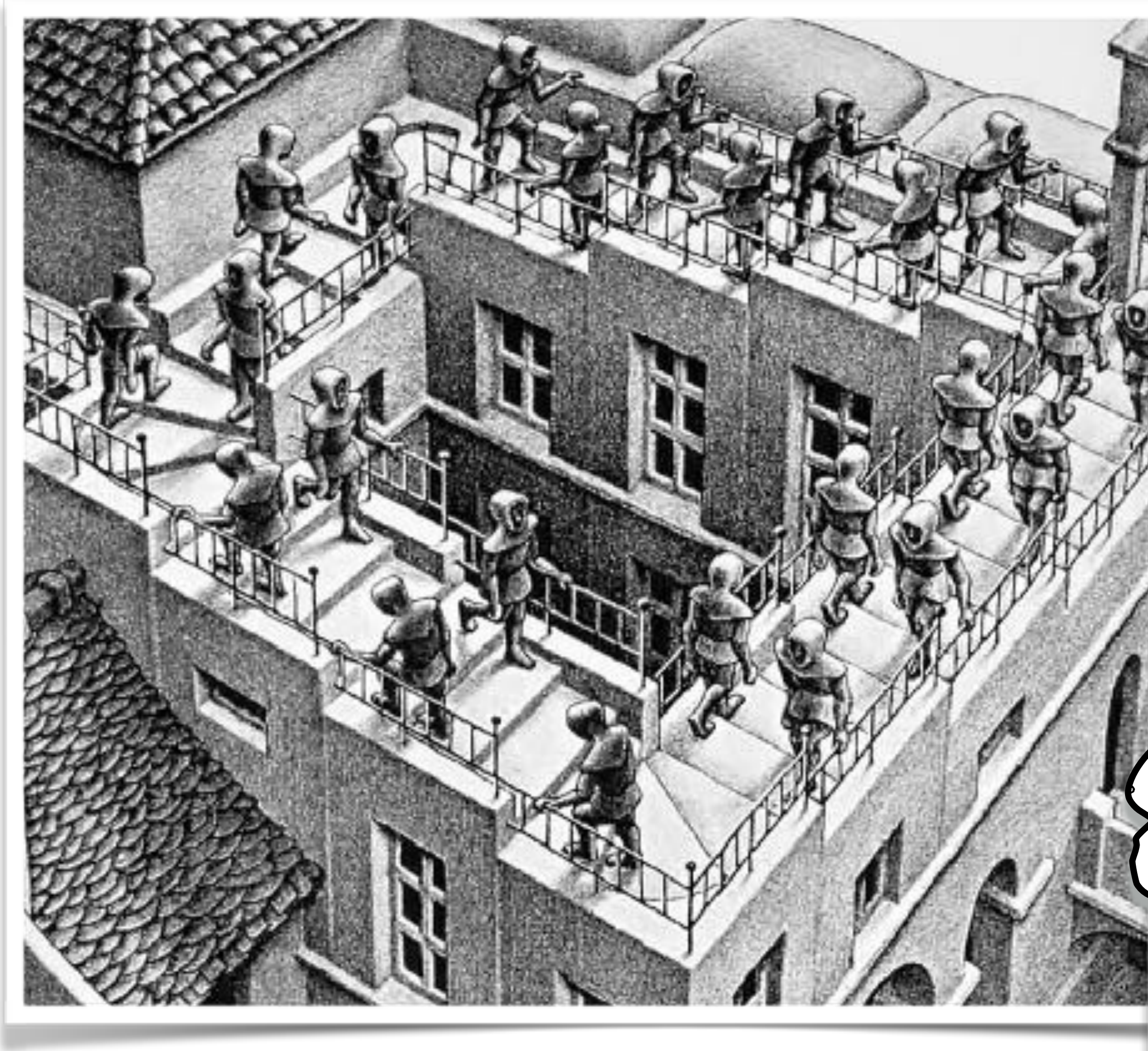
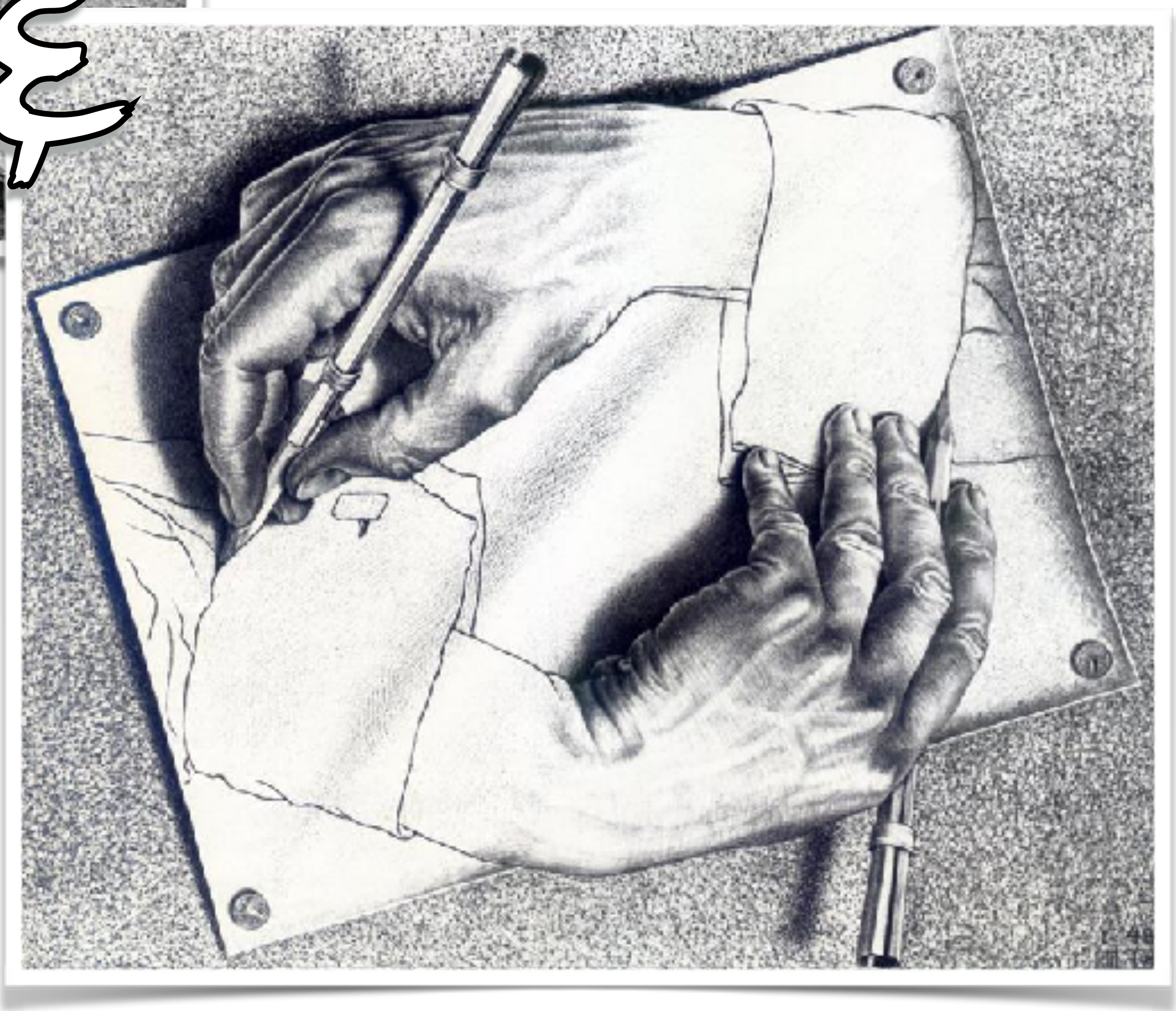


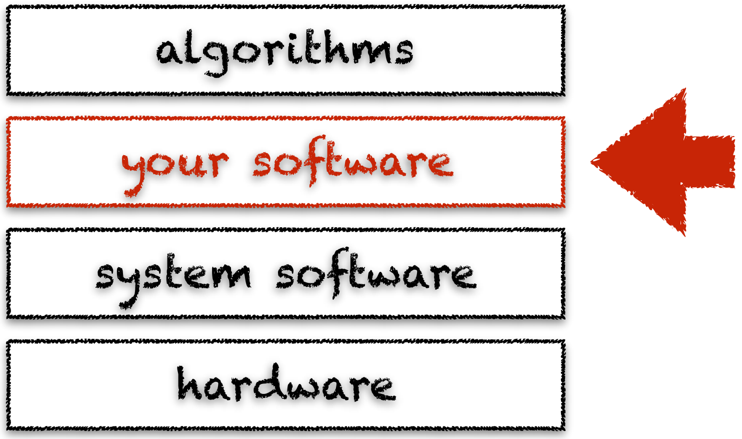
iteration

&



recursion

learning objectives



algorithms

your software

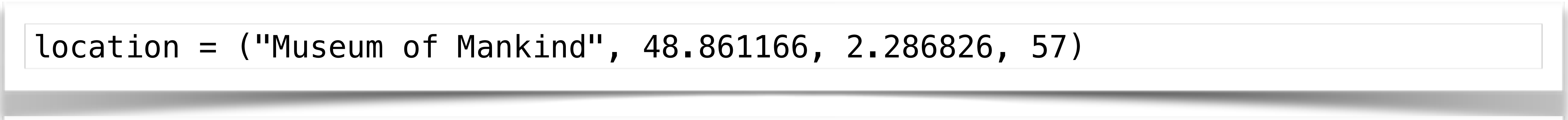
system software

hardware

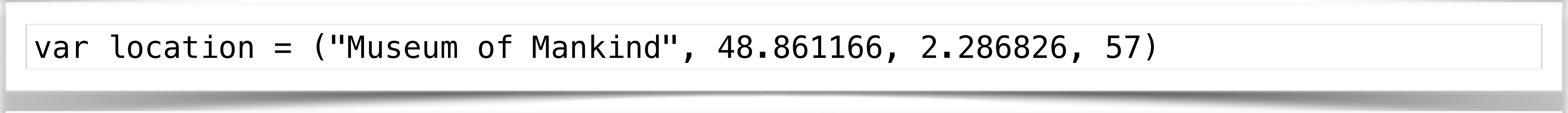
- learn about tuples, lists and maps
- learn about immutability and literals
- learn about iteration and recursion

notion of tuple

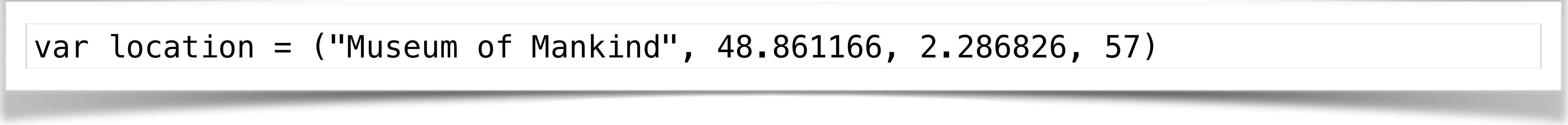
a tuple is finite ordered set of elements



```
location = ("Museum of Mankind", 48.861166, 2.286826, 57)
```



```
var location = ("Museum of Mankind", 48.861166, 2.286826, 57)
```



```
var location = ("Museum of Mankind", 48.861166, 2.286826, 57)
```

a n-tuple is an ordered set of n elements

- when n = 0 : we say it's an empty tuple or unit
- when n = 1 : we say it's single or singleton
- when n = 2 : we say it's double or couple or pair
- when n = 3 : we say it's triple or triplet or triad
- etc...

notion of tuple

accessing tuple elements

```
print("latitude is {0}, longitude is {1}, altitude is {2}m".format(location[1],location[2],location[3]))
```

```
print(s"latitude is ${location._2}, longitude is ${location._3}, altitude is ${location._4}m")
```

```
print("latitude is \(location.1), longitude is \(location.2), altitude is \(location.3)m")
```

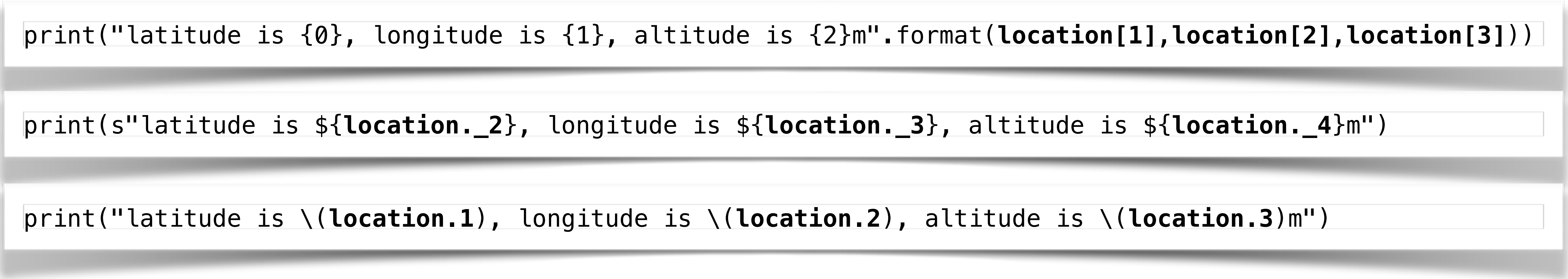
0	"Museum of Mankind”
1	48.861166
2	2.286826
3	57

1	"Museum of Mankind”
2	48.861166
3	2.286826
4	57

0	"Museum of Mankind”
1	48.861166
2	2.286826
3	57

notion of tuple

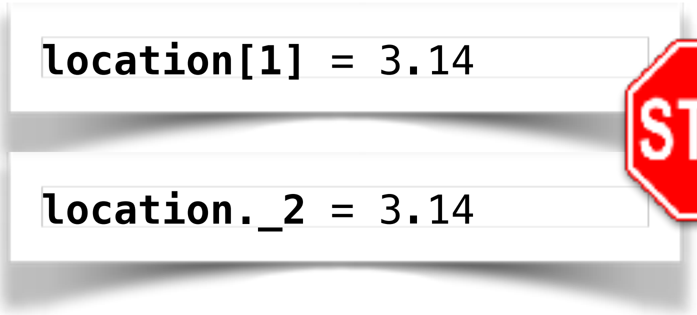
accessing tuple elements



```
print("latitude is {0}, longitude is {1}, altitude is {2}m".format(location[1],location[2],location[3]))
```

```
print(s"latitude is ${location._2}, longitude is ${location._3}, altitude is ${location._4}m")
```

```
print("latitude is \(location.1), longitude is \(location.2), altitude is \(location.3)m")
```

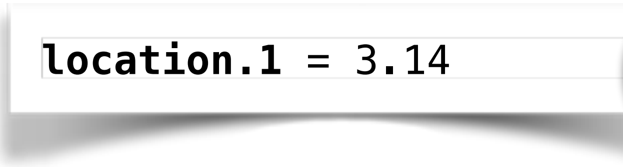


```
location[1] = 3.14
```

```
location._2 = 3.14
```

STOP

in scala and in python, tuples a immutable



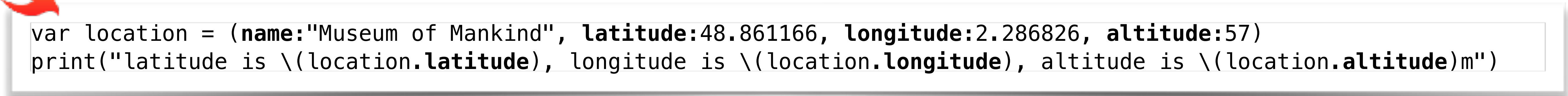
```
location.1 = 3.14
```

GO

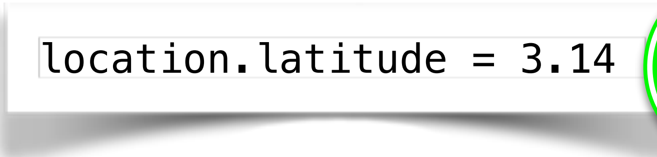
in swift, tuples are mutable ⇔ elements can be changed

notion of tuple

naming tuple elements

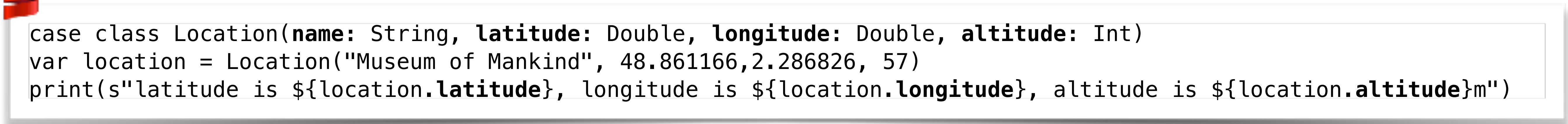


```
var location = (name:"Museum of Mankind", latitude:48.861166, longitude:2.286826, altitude:57)
print("latitude is \(location.latitude), longitude is \(location.longitude), altitude is \(location.altitude)m")
```

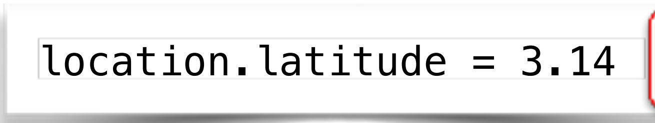


```
location.latitude = 3.14
```

GO



```
case class Location(name: String, latitude: Double, longitude: Double, altitude: Int)
var location = Location("Museum of Mankind", 48.861166,2.286826, 57)
print(s"latitude is ${location.latitude}, longitude is ${location.longitude}, altitude is ${location.altitude}m")
```



```
location.latitude = 3.14
```

STOP

named elements are not supported out-of-the box in Python tuples

immutability

an immutable object is an object whose state cannot be modified after its initialization

```
location.latitude = 3.14
```

STOP

an mutable object is an object whose state can be modified after its initialization

```
location.latitude = 3.14
```

GO

immutable objects are easier to share across your code because they are immune to side effects

in addition, the compiler (or the interpreter) can perform optimization on immutable objects

collections

many programs rely on collections of objects

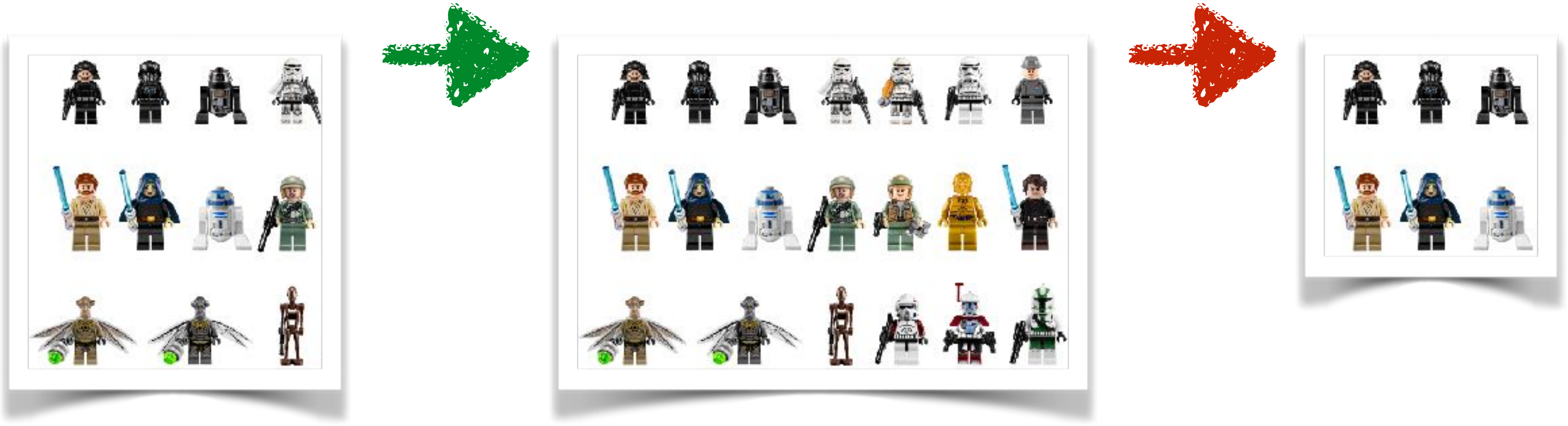
game elements

library catalog

notes in a notebook

collections

the number of items stored in a collection may vary over time



items added

items deleted

list creation & access

```
tour = ["Museum of Mankind", "Eiffel Tower", "Champs Elysée"]
```

```
var tour = List("Museum of Mankind", "Eiffel Tower", "Champs Elysée")
```

```
var tour = ["Museum of Mankind", "Eiffel Tower", "Champs Elysée"]
```

```
print(tour[1])
```

→ **Eiffel Tower**

```
print(tour(1))
```

→ **Eiffel Tower**

```
print(tour[1])
```

→ **Eiffel Tower**

literals

in a program, a literal is a notation for representing a value directly in the source code

	python	scala	swift
string	"Museum of Mankind" 'Museum of Mankind'	"Museum of Mankind"	
double	3.14		
float		3.14f	
integer	666		
boolean	True / False	true / false	true / false
tuple	("Museum of Mankind", 48.861166, 2.286826, 57)		
list	["one", "two", "three"]	List("one", "two", "three")	["one", "two", "three"]

adding & removing elements from a list

append	prepend
`tour.append("Triumphal Arch")`	`tour.insert(0,"Triumphal Arc")`
`tour = tour ::: List("Triumphal Arch")`	`tour = "Triumphal Arc"::tour`
`tour.append("Triumphal Arch")`	`tour.insert("Triumphal Arch”, at:0)`

remove first element	remove last element
`del tour[0]`	`tour.pop()`
`tour = tour.tail`	`tour = tour.take(tour.size – 1)`
`tour.remove(at:0)`	`tour.remove(at:tour.count – 1)`

adding & removing elements from a list

in scala, lists are immutable, so we have to create a new list for each modification

```
tour = tour ::: List("Triumphal Arch")
```

```
tour = "Triumphal Arc"::tour
```

```
tour = tour.tail
```

```
tour = tour.take(tour.size – 1)
```

if you need a mutable list, use a ListBuffer

```
import scala.collection.mutable.ListBuffer

var tour = ListBuffer("Museum of Mankind", "Eiffel Tower")    ➡ ("Museum of Mankind", "Eiffel Tower")

tour.append("Triumphal Arch")                                  ➡ ("Museum of Mankind", "Eiffel Tower", "Triumphal Arch")

tour.remove(0)                                                 ➡ ("Eiffel Tower", "Triumphal Arch")

tour.prepend("Champs Elysée")                                  ➡ ("Champs Elysée", "Eiffel Tower", "Triumphal Arch")

tour.trimEnd(1)                                                ➡ ("Champs Elysée", "Eiffel Tower")
```

adding & removing elements from a list

in swift, a lists is **mutable**, if and only if we are **accessing it via a variable**

```
var tour = ["Museum of Mankind", "Eiffel Tower", "Champs Elysée"]
tour.append("Triumphal Arch")
tour.remove(at:0)
```

GO

```
let tour = ["Museum of Mankind", "Eiffel Tower", "Champs Elysée"]
tour.append("Triumphal Arch")
tour.remove(at:0)
```

STOP

```
var myTour = ["Museum of Mankind", "Eiffel Tower", "Champs Elysée"]
myTour.append("Triumphal Arch")
myTour.remove(at:0)
```

GO

```
let yourTour = myTour
yourTour.append("Triumphal Arch")
yourTour.remove(at:0)
```

STOP

associative arrays

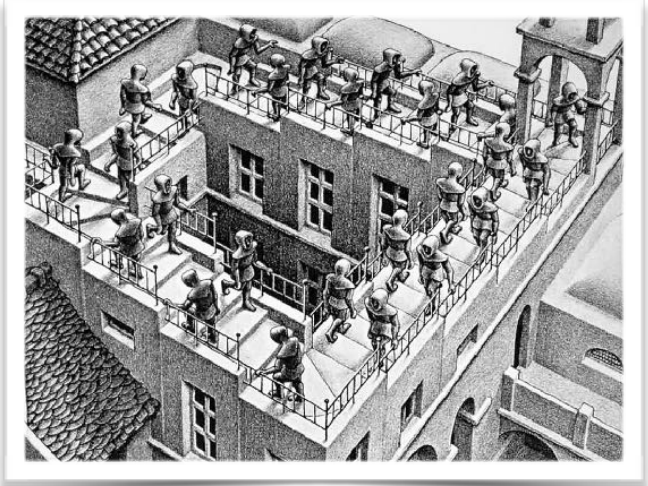
in a program, an **associative array** (also called a **dictionary** or simply a **map**) is a collection composed of a set of **(key, value) pairs**, where **each key appears at most once** in the collection

```
mountains = {"jungfrau": 4158, "eiger": 0}    ➡  {'eiger': 0, 'jungfrau': 4158}
height = mountains["eiger"]                   ➡  0
mountains["eiger"] = 3950                     ➡  {'eiger': 3950, 'jungfrau': 4158}
mountains["moench"] = 4099                    ➡  {'eiger': 3950, 'jungfrau': 4158, 'moench': 4099}
mountains.pop("jungfrau")                     ➡  {'eiger': 3950, 'moench': 4099}
```

```
var mountains = scala.collection.mutable.Map("jungfrau" -> 4158, "eiger" -> 0)
                                              ➡  Map(jungfrau -> 4158, eiger -> 0)
var height = mountains("eiger")               ➡  0
mountains("eiger") = 3950                     ➡  Map(jungfrau -> 4158, eiger -> 3950)
mountains("moench") = 4099                    ➡  Map(jungfrau -> 4158, eiger -> 3950, moench -> 4099)
mountains.remove("jungfrau")                  ➡  Map(eiger -> 3950, moench -> 4099)
```

```
var mountains = ["jungfrau": 4158, "eiger": 0]  ➡  ["eiger": 0, "jungfrau": 4158]
height = mountains["eiger"]                     ➡  0
mountains["eiger"] = 3950                       ➡  ["eiger": 3950, "jungfrau": 4158]
mountains["moench"] = 4099                      ➡  ["moench": 4099, "eiger": 3950, "jungfrau": 4158]
mountains.removeValue(forKey:"jungfrau")        ➡  ["eiger": 3950, "moench": 4099]
```

iteration



we often want to perform some actions an arbitrary number of times e.g.,

print all the notes in a notebook

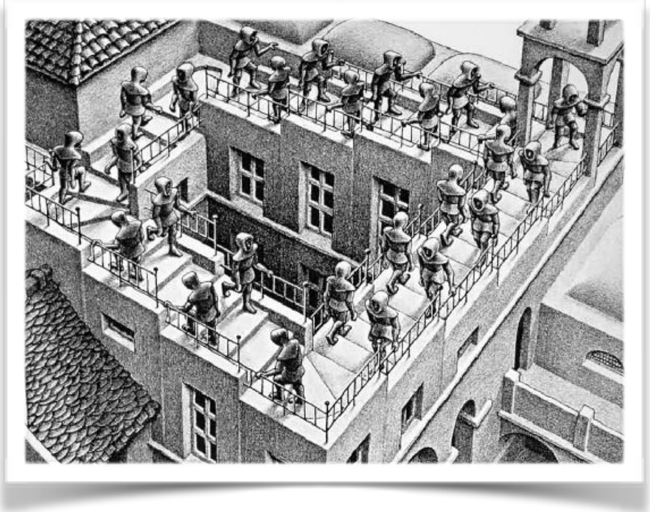
convert the height of a mountains from meters to feet

compute the list of 100 first prime numbers in sequence

with collections in particular, we often want to repeat a sequence of actions once for each object in a given collection

programming languages include loop statements for this

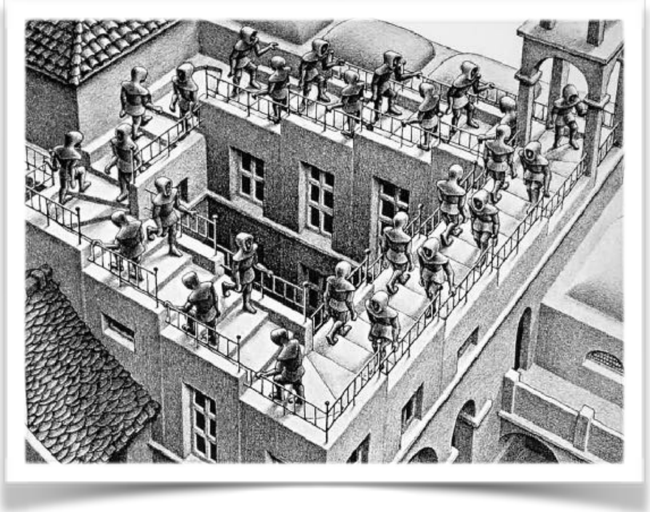
iteration



for each loop

while loop

for each loop



a for-each loop repeats the loop body for each and every object in a collection

for each loop

python

iterating through a list

```
mountains = { "jungfrau", "eiger", "moench"}
for summit in mountains:
    print("I will climb to the summit of the {0}".format(summit))
```

iterating through a map

```
mountains = { "jungfrau":4158, "eiger":3950, "moench":4099}
height = 0
for summit in mountains.keys():
    print("I will climb to the summit of the {0} at {1} meters".format(summit,mountains[summit]))
    height = height + mountains[summit]
print("In total, I will climb {0} meters".format(height))
```

scala

iterating through a list

```
var mountains = List("jungfrau", "eiger", "moench")
for(summit <- mountains)
  println(s"I will climb to the summit of the $summit")
```

iterating through a map

```
var mountains = Map("jungfrau"->4158, "eiger"->3950, "moench"->4099)
var height = 0
for (summit <- mountains.keys) {
  println(s"I will climb to the summit of the $summit at ${mountains(summit)} meters");
  height = height + mountains(summit)
}
println(s"In total, I will climb $height meters")
```

swift

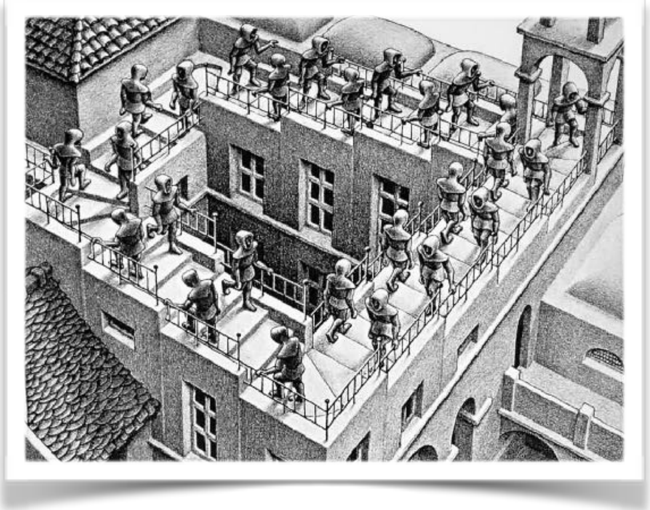
iterating through a list

```
var mountains = ["jungfrau", "eiger", "moench"]
for summit in mountains {
    print("I will climb to the summit of the \(summit)")
}
```

iterating through a map

```
var mountains = ["jungfrau":4158, "eiger":3950, "moench":4099]
var height = 0
for summit in mountains.keys {
    print("I will climb to the summit of the \(summit) at \(mountains[summit]!) meters")
    height = height + mountains[summit]!
}
print("In total, I will climb \(height) meters")
```

while loop



a while loop uses a boolean condition to decide whether or not to continue the loop

while loop

python

```
numbers = [1,2,4,8,16,32,64, 128,256]
sum = 0
i = 0
while sum < 512 and i < len(numbers):
    sum = sum + numbers[i]
    i = i + 1
print("the sum is {0}".format(sum))
```

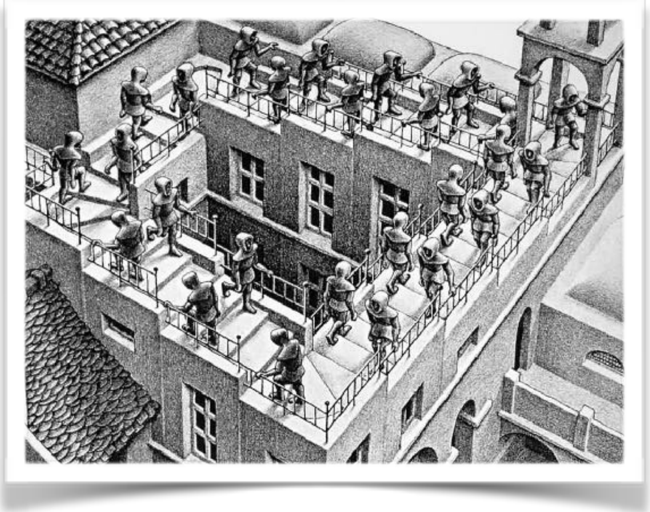
scala

```
var numbers = List(1, 2, 4, 8, 16, 32, 64, 128, 256)
var sum = 0
var i = 0
while (sum < 512 && i < numbers.length) {
  sum = sum + numbers(i)
  i = i + 1
}
print(s"the sum is $sum")
```

swift

```
var numbers = [1, 2, 4, 8, 16, 32, 64, 128, 256]
var sum = 0
var i = 0
while (sum < 512 && i < numbers.count) {
    sum = sum + numbers[i]
    i = i + 1
}
print("the sum is \(sum)")
```

iteration



for-each

simpler: it is easier to write

safer: it is guaranteed to stop

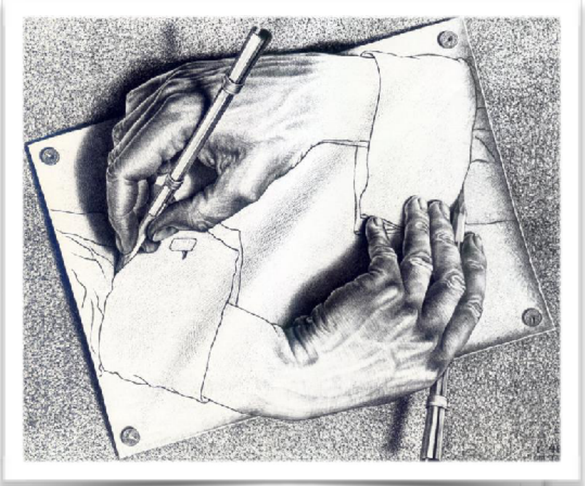
while

efficient: can process part of a collection

versatile: can be used for other purposes

be careful: could be an infinite loop

recursion



a classical way to solve a problem is to divide it into smaller and easier subproblems

if one of the subproblems is a less complex instance of the original problem, you might want to consider using recursion

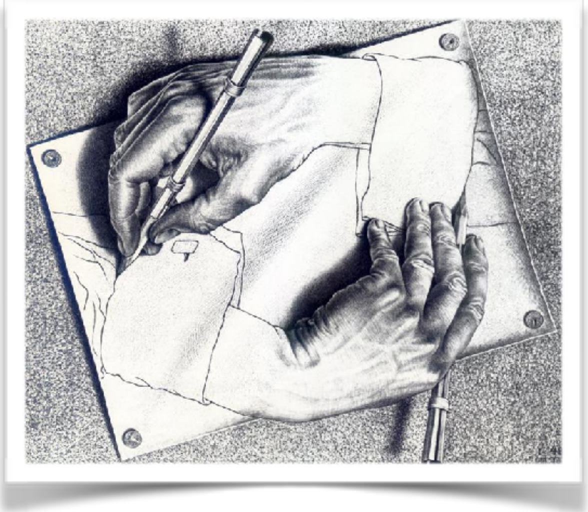
for example, the factorial of n can be defined as

$$n! = 1 \times 2 \times 3 \times .... \times (n-1) \times n$$

but it can also be defined as:

$$n! = (n - 1)! \times n$$

recursion
fibonacci numbers

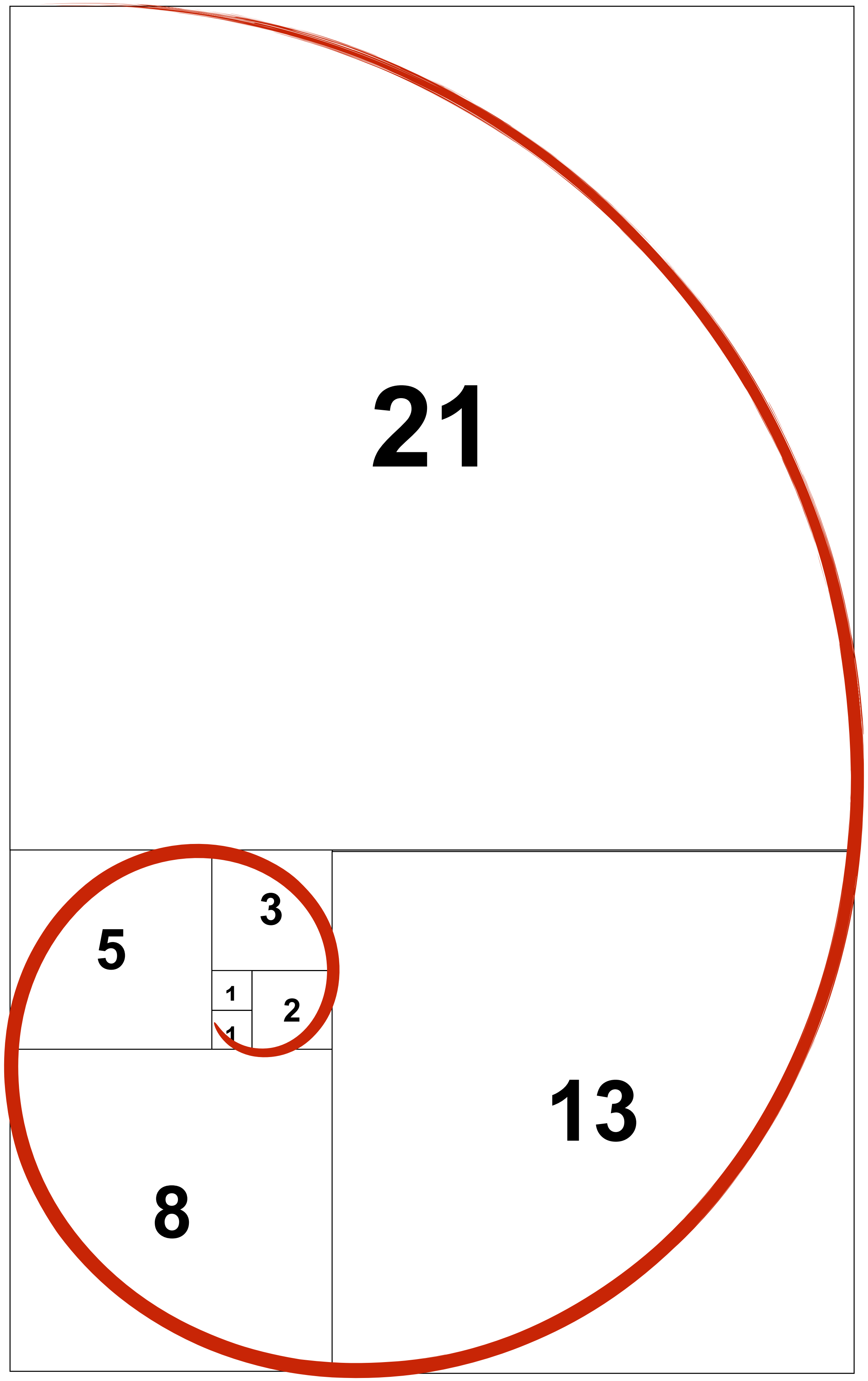


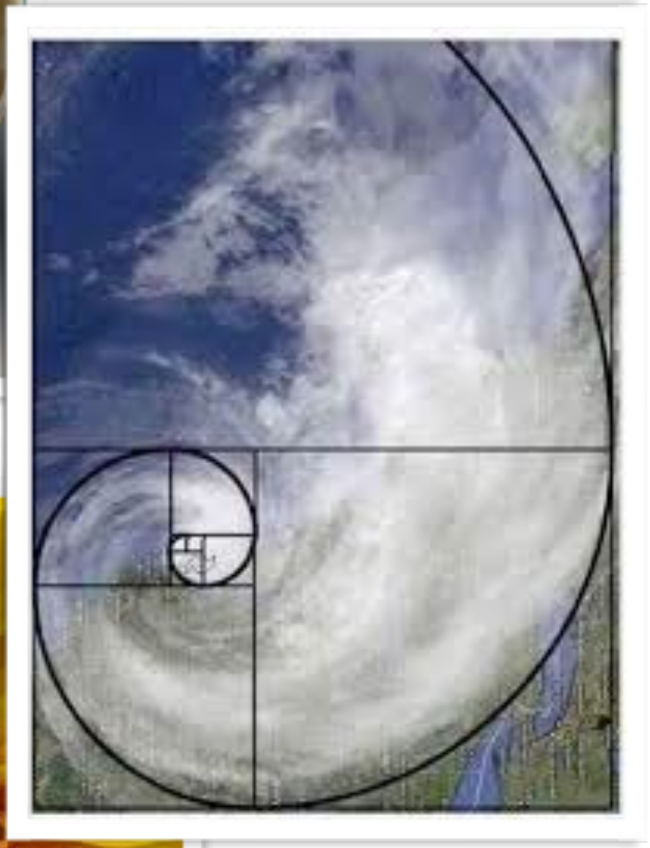
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	...

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	159			

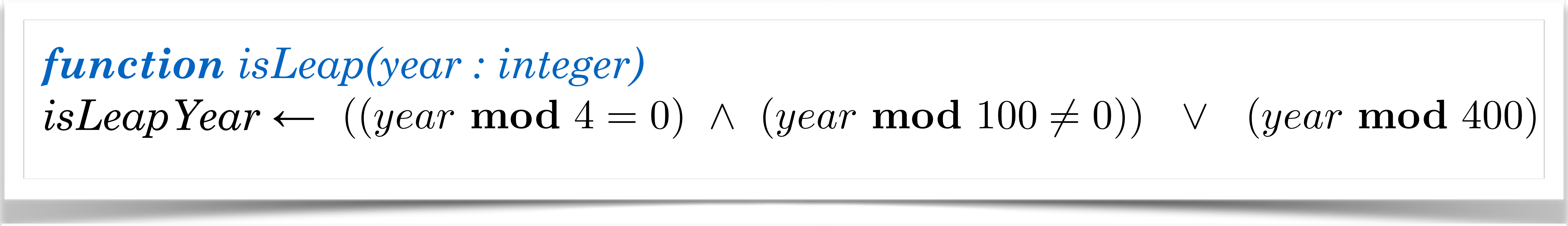
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$



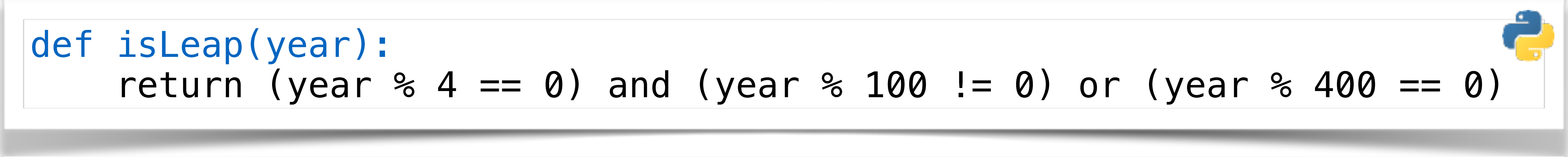


what's a function?

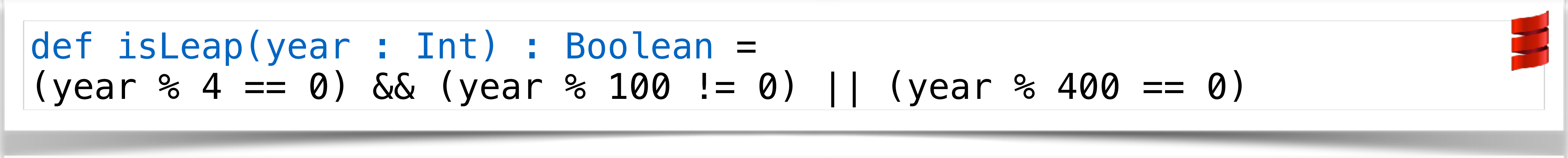
we already (quietly) introduced the notion of function in the previous lesson



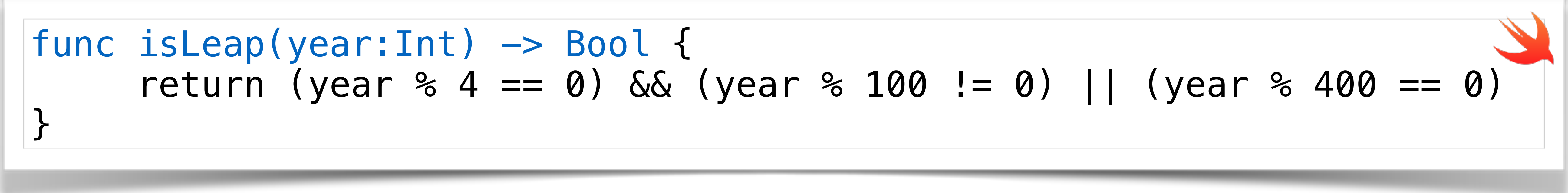
```
$\textbf{\textit{function}}\ isLeap(year : integer)$
$isLeapYear \leftarrow ((year \textbf{ mod } 4 = 0) \land (year \textbf{ mod } 100 \neq 0)) \lor (year \textbf{ mod } 400)$
```



```
def isLeap(year):
    return (year % 4 == 0) and (year % 100 != 0) or (year % 400 == 0)
```



```
def isLeap(year : Int) : Boolean =
(year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
```



```
func isLeap(year:Int) -> Bool {
     return (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
}
```

what's a function?

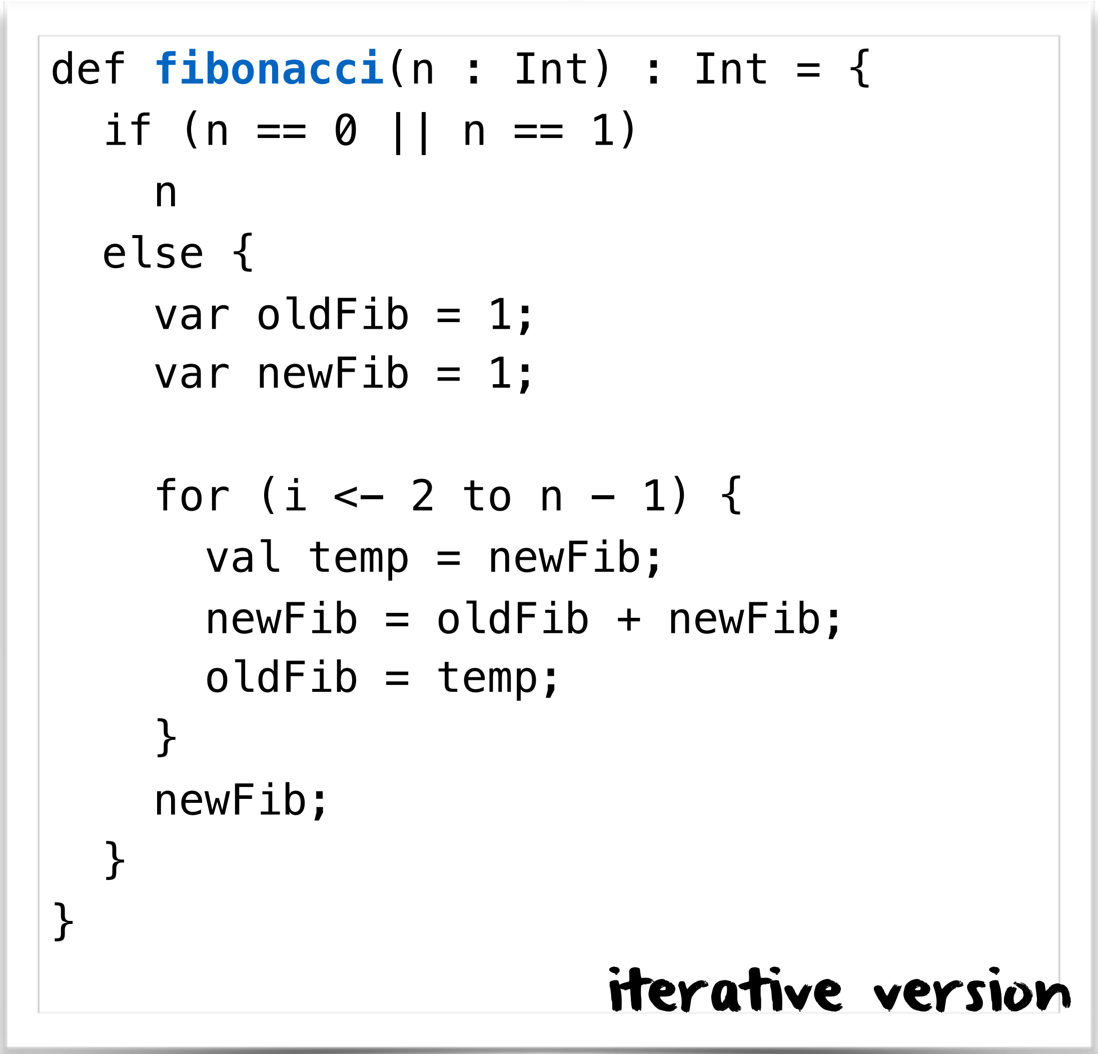
in a program, a function is a sequence of instructions performing a specific task, packaged as a reusable unit

depending on the context, a function is also sometimes called a subroutine, a procedure or a method

fibonacci numbers

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

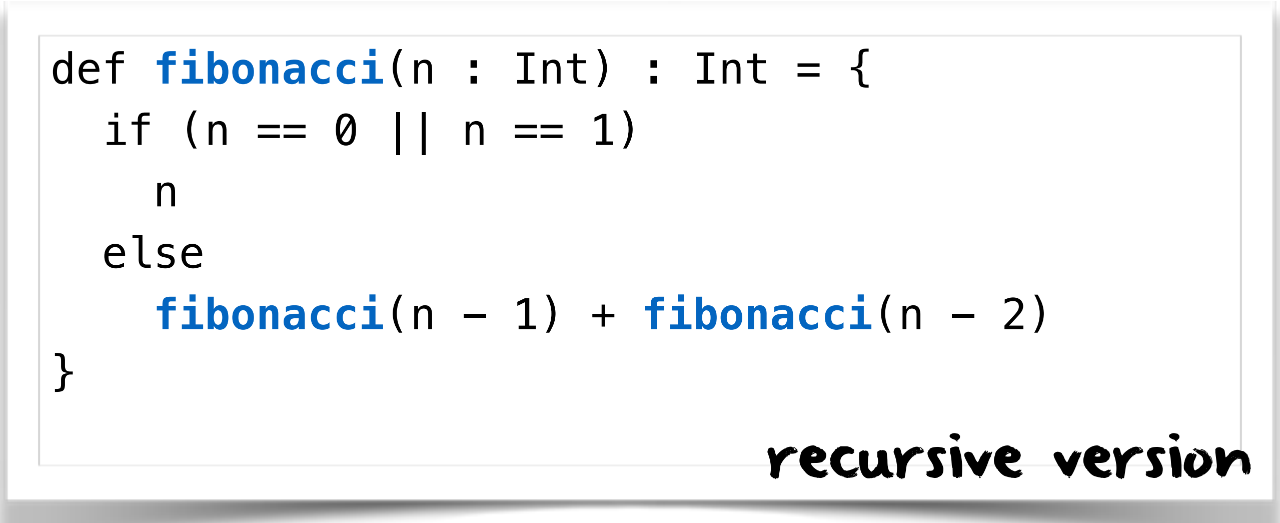
n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	...



```scala
def fibonacci(n : Int) : Int = {
  if (n == 0 || n == 1)
    n
  else {
    var oldFib = 1;
    var newFib = 1;

    for (i <- 2 to n - 1) {
      val temp = newFib;
      newFib = oldFib + newFib;
      oldFib = temp;
    }
    newFib;
  }
}
```

iterative version



```scala
def fibonacci(n : Int) : Int = {
  if (n == 0 || n == 1)
    n
  else
    fibonacci(n - 1) + fibonacci(n - 2)
}
```

recursive version

examples given in scala

function calls

breakpoint here

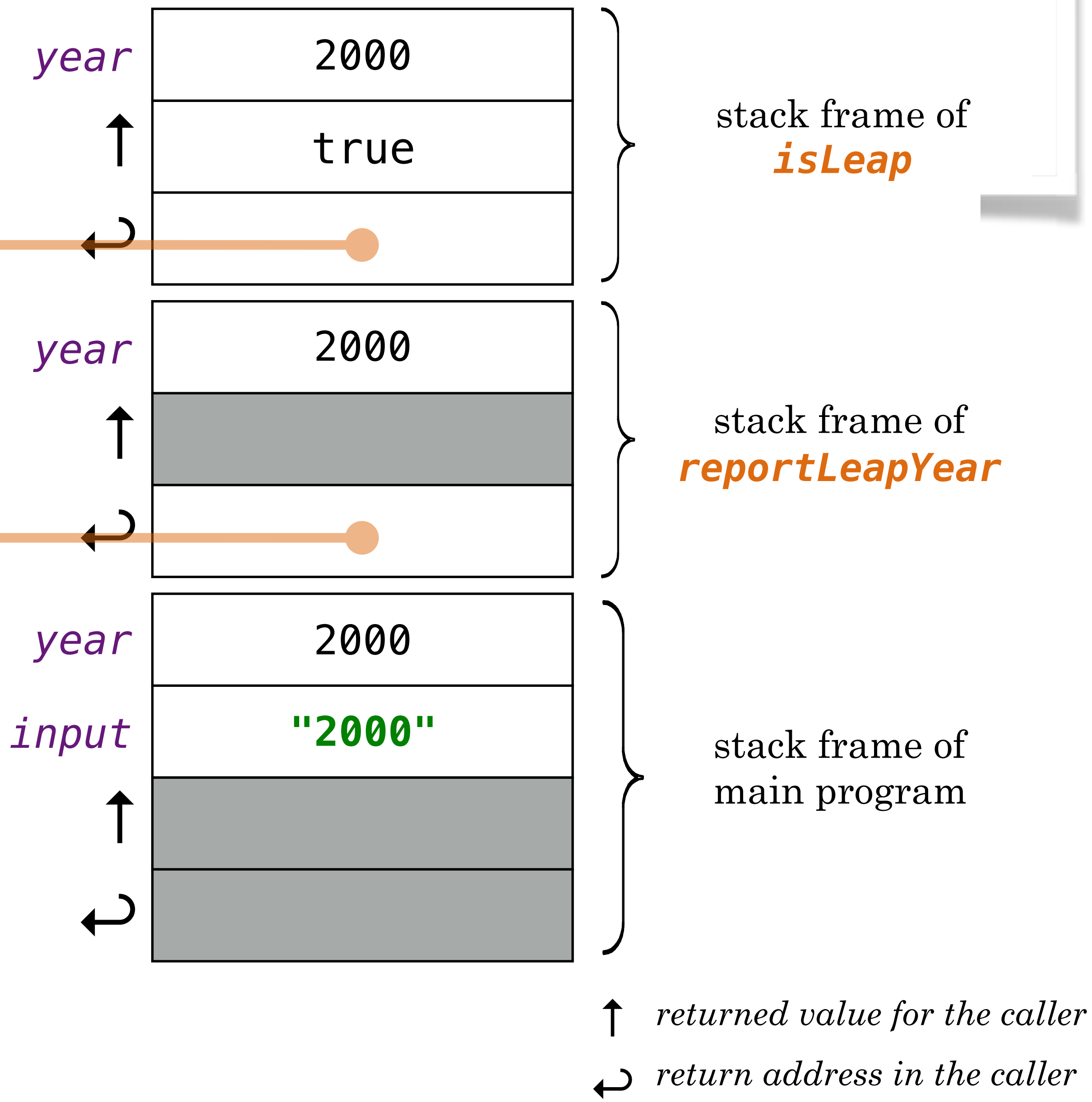
```scala
import scala.io.StdIn.readLine

object LeapYear extends App {

  def isLeap(year : Int) : Boolean = (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)

  def reportLeapYear(year : Int) = {
    print(s"Is $year a leap year? ${if (isLeap(year))  "Yes, it is!" else "No, it's not!"}")
  }

  val year = readLine("Give us a year! ").toInt
  reportLeapYear(year)
}
```



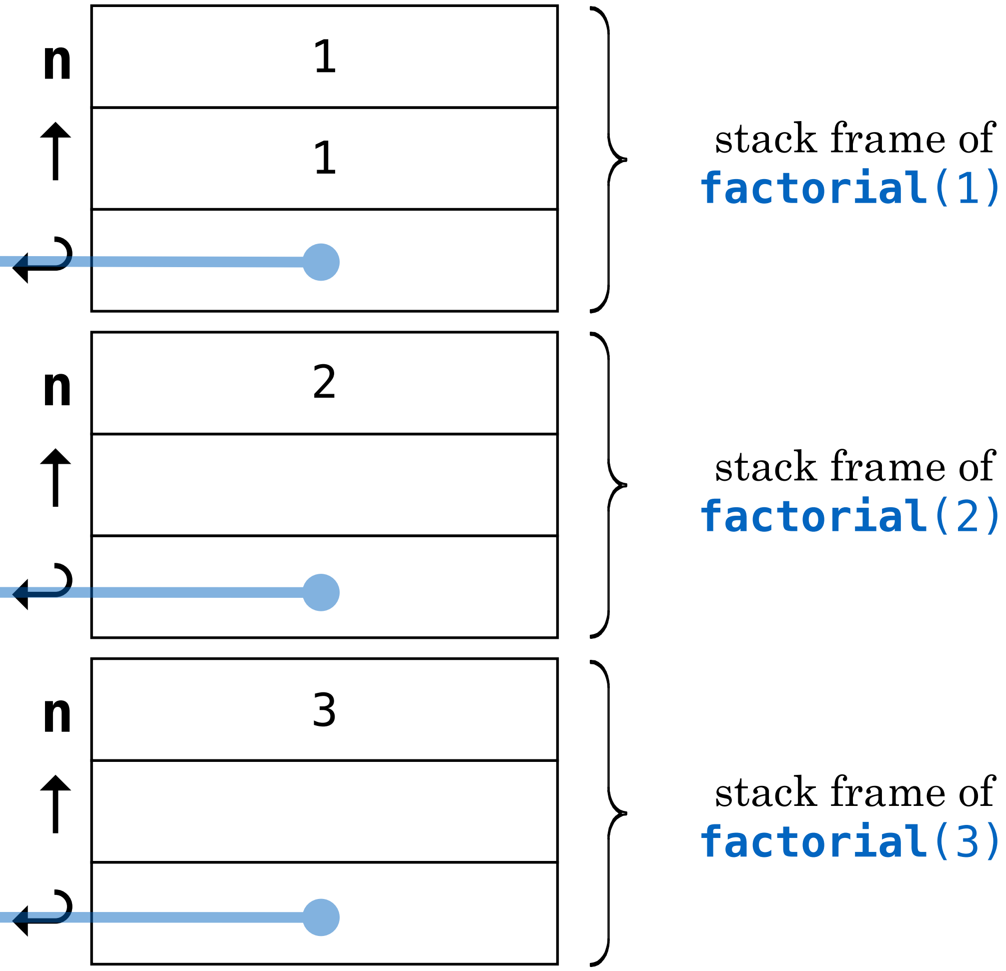
the call stack contains a stack frame for each function call currently active

a stack frame contains all the local variables and parameters of the function being called

recursive function calls

```
def factorial(n: Int) : Int = {
  if (n == 0 || n == 1) {
    1
  }
  else {
    factorial(n-1) * n
  }
}
```

there must be
a stop condition
for the recursion



after calling factorial(3), we have the following execution stack when the stop condition is reached:

↑ *returned value for the caller*

↩ *return address in the caller*

question

what happens if we pass n = -1 ?