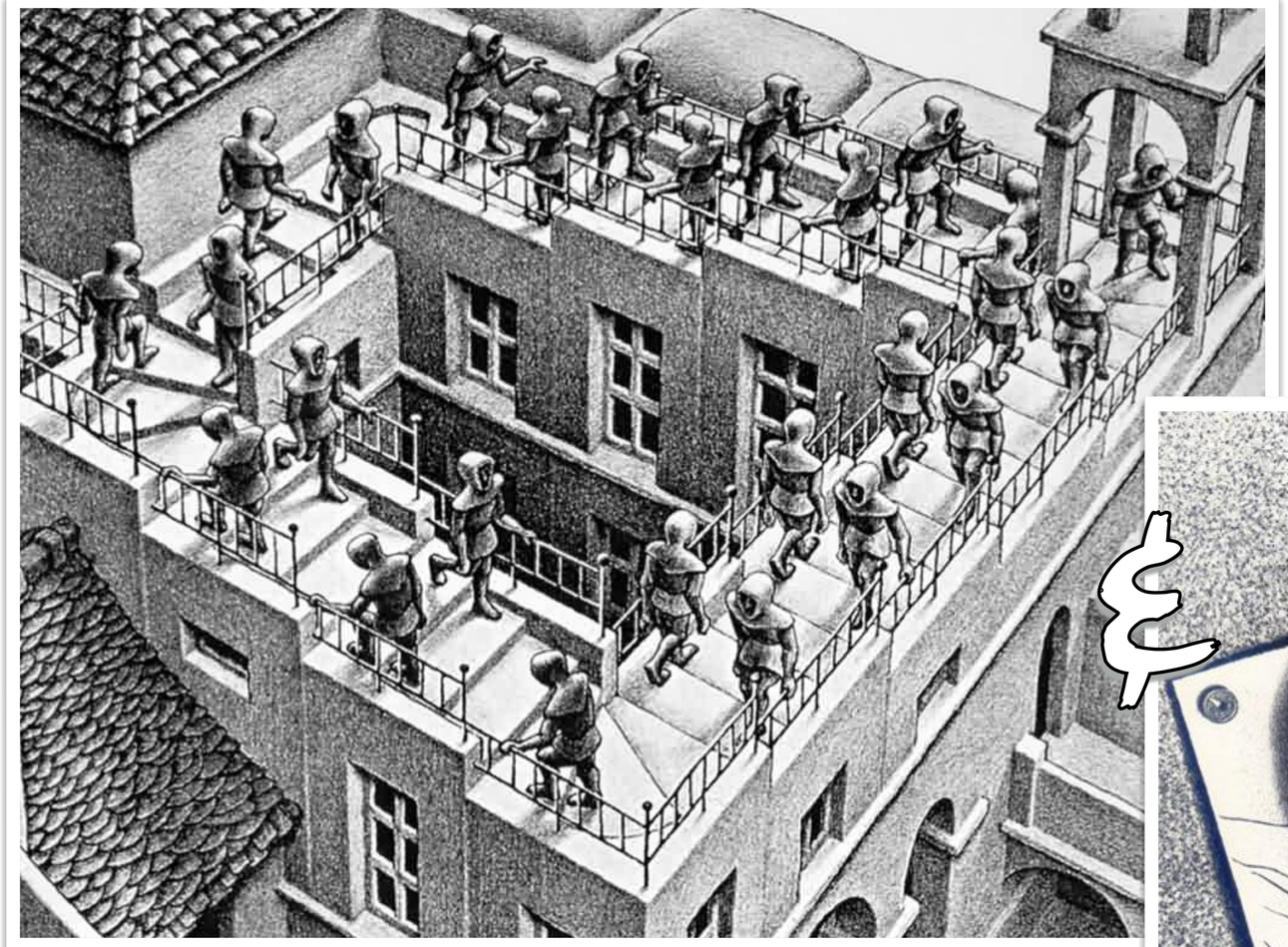
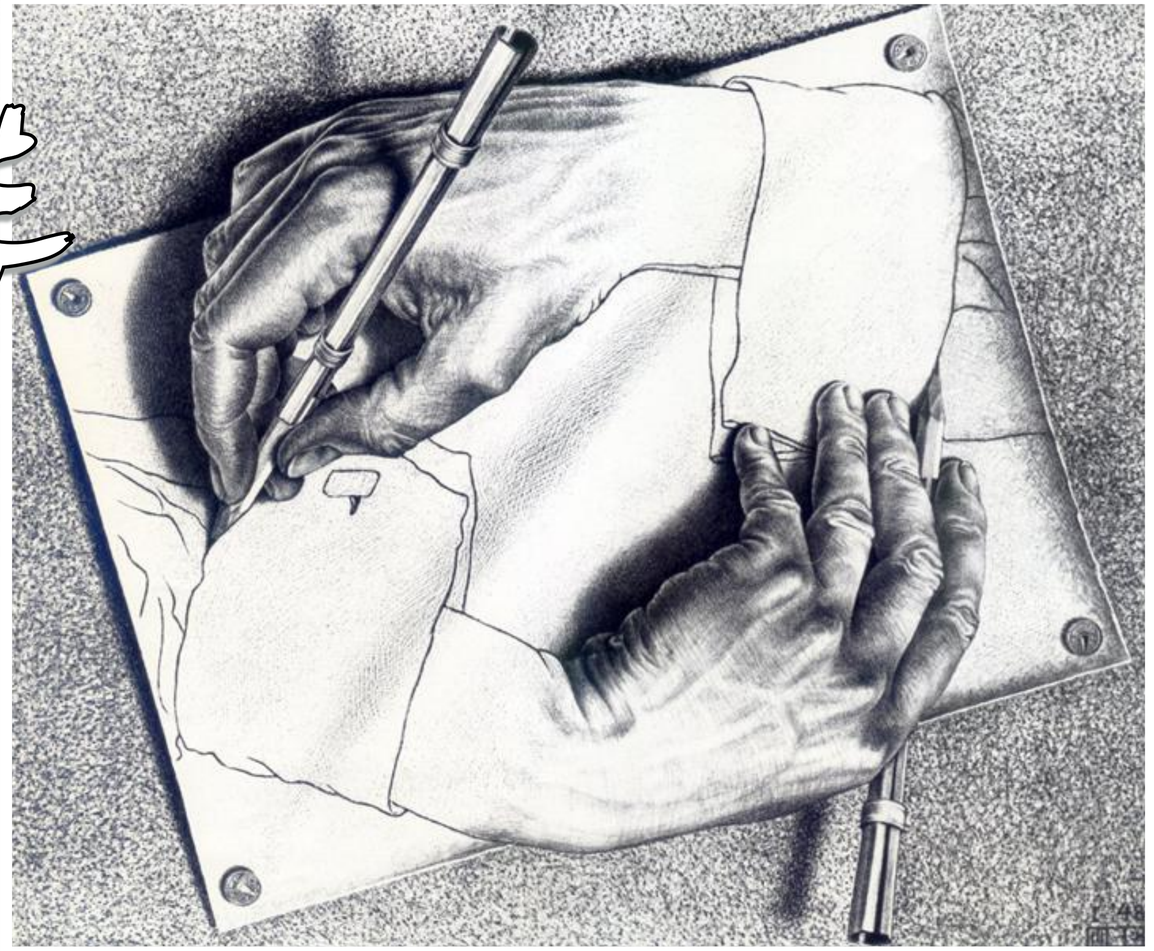
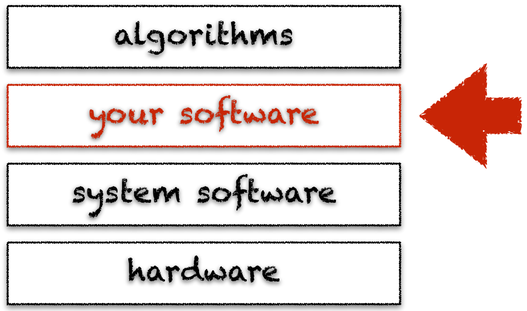


recursion

iteration



learning objectives



- 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)
```

```
Object[] location = {"Museum of Mankind", 48.861166, 2.286826, 57};
```

an array, really

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

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

```java
System.out.println("latitude is "  + location[1] + ", " +
                   "longitude is " + location[2] + ", " +
                   "altitude is "  + location[3]);
```

'+' is the string concatenation operator

0	"Museum of Mankind"
1	48.861166
2	2.286826
3	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]))
```

```
System.out.println("latitude is "  + location[1] + ", " +
                   "longitude is " + location[2] + ", " +
                   "altitude is "  + location[3]);
```

'+' is the string concatenation operator

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

in python, tuples
are immutable

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

in java, tuples are mutable
⇔ they can be changed

immutability

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


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

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

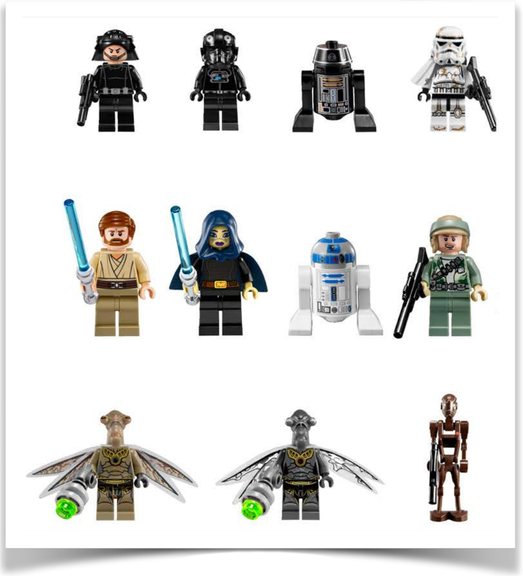

```
location[1] = 3.14;
```

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

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

```java
List<String> tour = Arrays.asList("Museum of Mankind", "Eiffel Tower", "Champs Elysée");
```

Java 8

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

since Java 9

```python
print(len(tour))
```
→ 3

```java
System.out.println(tour.size());
```
→ 3


```python
print(tour[1])
```
→ Eiffel Tower

```java
System.out.println(tour.get(1));
```
→ Eiffel Tower

literals

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

	python	java
string	`"Museum of Mankind"` `'Museum of Mankind'`	`"Museum of Mankind"`
double	`3.14`	
float		`3.14f`
integer	`666`	
boolean	`True / False`	`true / false`
tuple	`("Museum of Mankind", 48.861166, 2.286826, 57)`	`{"Museum", 48.86, 2.28, 57}`
list	`["Jan", "Feb", "Mar"]`	`Arrays.asList("Jan", "Feb", "Mar")` `List.of("Jan", "Feb", "Mar")`

Java 8

since Java 9

adding & removing elements from a list

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

append

```
tour.append("Triumphal Arch")
```

prepend

```
tour.insert(0,"Triumphal Arc")
```

remove first element

```
del tour[0]
```

remove last element

```
tour.pop()
```

adding & removing elements from a list

in java, lists are also **immutable** by default,
so we have to create a **mutable array list** or a
mutable linked list if we need to modify its content

Java 8

```
List<String> immutableTour = Arrays.asList("Museum of Mankind", "Eiffel Tower", "Champs Elysée");
List<String> tour = new ArrayList(immutableTour);
```

append

```
tour.add("Triumphal Arc");
```

prepend

```
tour.add(0, "Triumphal Arc");
```

remove first element

```
tour.remove(0);
```

remove last element

```
tour.remove(tour.size() - 1);
```

adding & removing elements from a list

in java, lists are also **immutable** by default,
so we have to create a **mutable array list** or a
mutable linked list if we need to modify its content

since Java 9

```
var immutableTour = List.of("Museum of Mankind", "Eiffel Tower", "Champs Elysée");
var tour = new LinkedList(immutableTour);
```

append

```
tour.addLast("Triumphal Arch");
```

prepend

```
tour.addFirst("Triumphal Arch");
```

remove first element

```
tour.removeFirst("Triumphal Arch");
```

remove last element

```
tour.removeLast("Triumphal Arch");
```

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 = new HashMap(Map.of("jungfrau", 4158, "eiger", 0));
                                                    ➡ {eiger=0, jungfrau=4158}
var height = mountains.get("eiger");                ➡ 0
mountains.put("eiger", 3950);                       ➡ {eiger=3950, jungfrau=4158}
mountains.put("moench", 4099);                      ➡ {eiger=3950, jungfrau=4158, moench=4099}
mountains.remove("jungfrau");                       ➡ {eiger=3950, moench=4099}
```

since Java 9

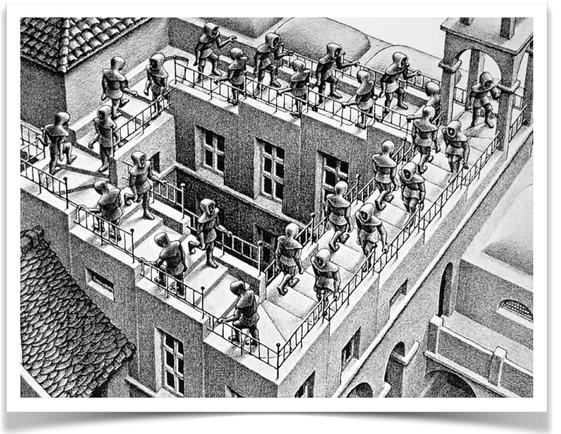
associative arrays

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

Java 8

```
Map<String, Integer> mountains = new HashMap<String, Integer>();
mountains.put("jungfrau", 4158);
mountains.put("eiger", 0);

                                                    ➥ {eiger=0, jungfrau=4158}
var height = mountains.get("eiger");                ➥ 0
mountains.put("eiger", 3950);                       ➥ {eiger=3950, jungfrau=4158}
mountains.put("moench", 4099);                      ➥ {eiger=3950, jungfrau=4158, moench=4099}
mountains.remove("jungfrau");                       ➥ {eiger=3950, moench=4099}
```

iteration



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

convert the height of a mountains from meters to feet

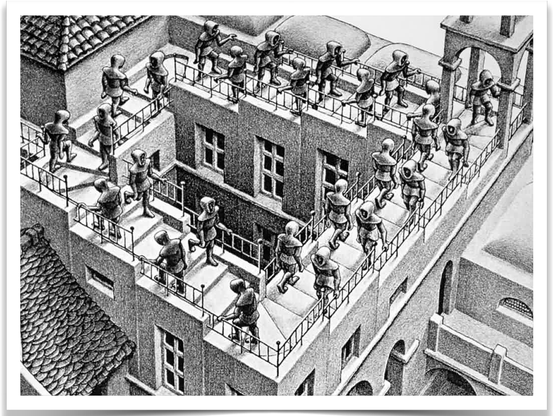
compute the list of 100 first prime numbers in sequence

print all the notes in a notebook

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

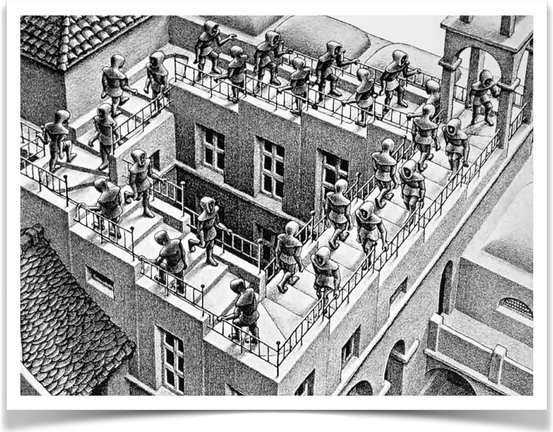
iteration



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))
```

java

since Java 9

iterating through a list

```
var mountains = List.of("jungfrau", "eiger", "moench");
for (var summit : mountains) {
  System.out.println("I will climb to the summit of the " + summit);
}
```

iterating through a map

```
var mountains = Map.of("jungfrau", 4158, "eiger", 3950, "moench", 4099);
var height = 0;
for (var summit : mountains.keySet()) {
  System.out.println("I will climb to the summit of the " + summit + " at " + mountains.get(summit) + " meters");
  height = height + mountains.get(summit);
}
System.out.println("In total, I will climb " + height +" meters");
```

for each loop

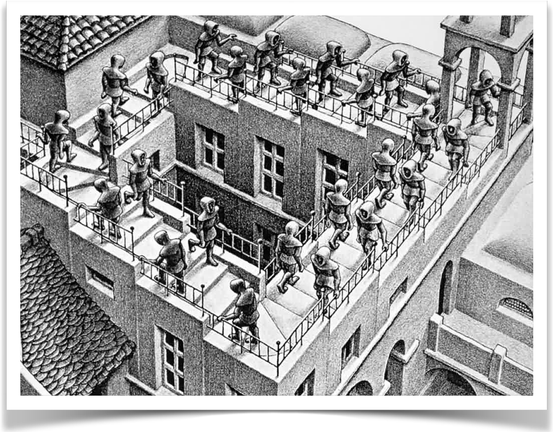
Java 8

	java

iterating through a list	`List<String> mountains = Arrays.asList("jungfrau", "eiger", "moench");`
	`for (String summit : mountains) {`
	`  System.out.println("I will climb to the summit of the " + summit);`
	`}`

iterating through a map	`Map<String,Integer> mountains = new HashMap();`
	`mountains.put("jungfrau", 4158); mountains.put("eiger", 3950); mountains.put("moench", 4099);`
	`int height = 0;`
	`for (String summit : mountains.keySet()) {`
	`  System.out.println("I will climb to the summit of the " + summit + " at " + mountains.get(summit) + " meters");`
	`  height = height + mountains.get(summit);`
	`}`

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))
```

java

since Java 9

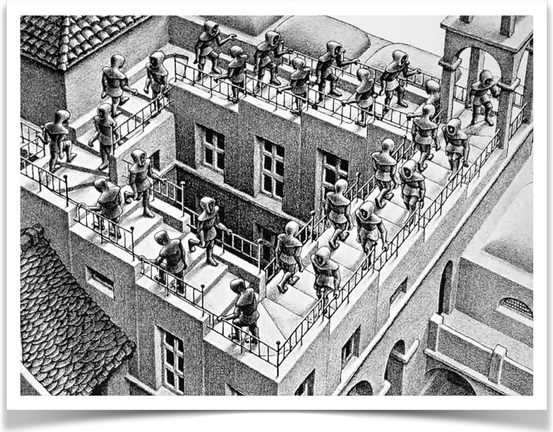
```
var numbers = List.of(1, 2, 4, 8, 16, 32, 64, 128, 256);
var sum = 0;
var i = 0;
while (sum < 512 && i < numbers.size()) {
    sum = sum + numbers.get(i);
    i = i + 1;
}
System.out.println("the sum is " + sum);
```

java

Java 8

```
List<Integer> numbers = Arrays.asList(1, 2, 4, 8, 16, 32, 64, 128, 256);
int sum = 0;
int  i = 0;
while (sum < 512 && i < numbers.size()) {
  sum = sum + numbers.get(i);
  i = i + 1;
}
System.out.println("the sum is " + sum);
```

iteration



for-each

simpler: easier to write

safer: guaranteed to stop

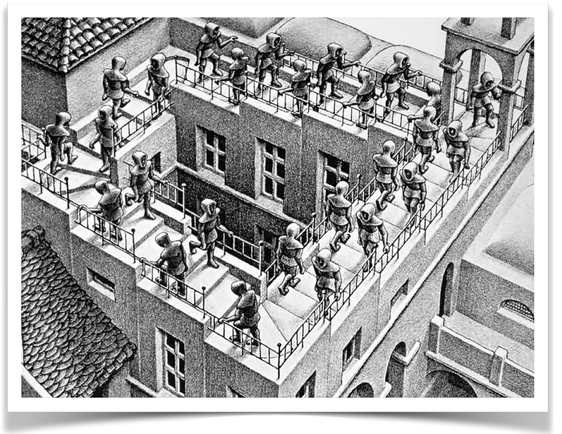
while

efficient: can process part of a collection

versatile: can be used for other purposes

be careful: could be an infinite loop

iterators



what happens if we try to **remove an element** from a list **while iterating**?

problem

```
List<Integer> numbers = new ArrayList(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8));
for (int x : numbers) {
  if (x < 6) {
    numbers.remove(x);
  }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
        at java.util.ArrayList$Itr.next(ArrayList.java:859)
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]

for x in numbers:
        if x < 6:
            numbers.remove(x);
print(numbers)
```

[2, 4, 6, 7, 8]

iterators in python do not provide a remove operation

😟

solution

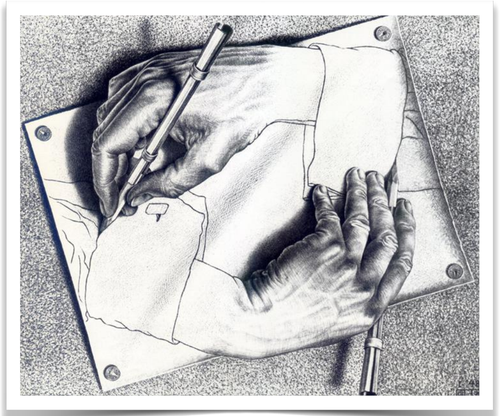
an **iterator** allows to traverse a collection, particularly lists, and to keep track of where we are in the iteration, even when items are removed on the fly

```
List<Integer> numbers = new ArrayList(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8));
Iterator<Integer> iter = numbers.iterator();

while(iter.hasNext()) {
  if (iter.next() < 6) {
    iter.remove();
  }
}
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]

for x in list(numbers):
        if x < 6:
            numbers.remove(x)
print(numbers)
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
numbers = [x for x in numbers if x >= 6]
```

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

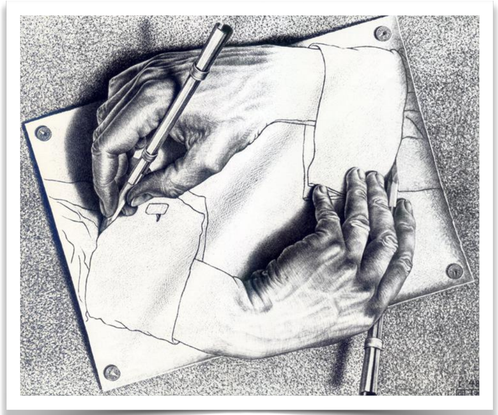
for example, the factorial of n can be defined as

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

but it can also be defined as:

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

recursion
factorial



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

but it can also be defined as:

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

```
def factorial(n):
    if n <= 1:
        return 1
    f = 1
    for i in range(2,n+1):
        f = f*i
    return f
```

iterative version

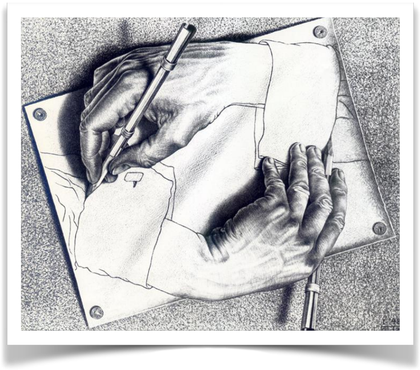
```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

recursive version

example given in python

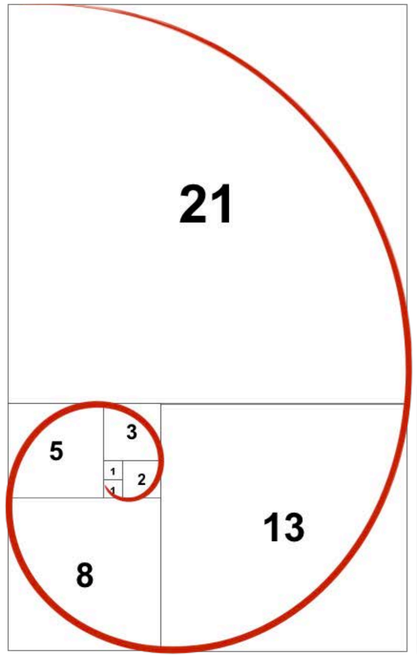
recursion

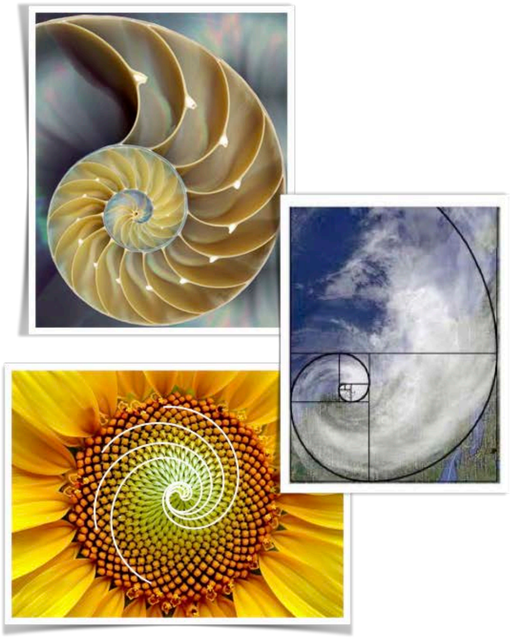
fibonacci numbers



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

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	...





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	...

```java
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return n;
  } else {
    int oldFib = 1;
    int newFib = 1;
    for (int i = 2; i < n; i++) {
      int temp = newFib;
      newFib = oldFib + newFib;
      oldFib = temp;
    }
    return newFib;
  }
}
```

iterative version

```java
int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return n;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

recursive version

example given in java

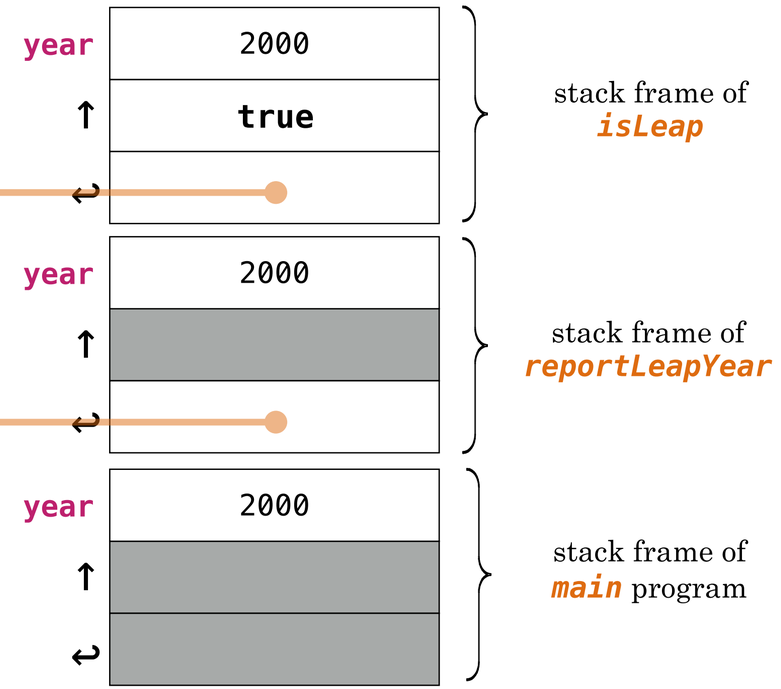
call stack - principle

```
public class LeapYear {

  static boolean isLeap(int year) {
    return (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0);
  }

  static void reportLeapYear(int year) {
    System.out.println("Is " + year + " a leap year?" +  (isLeap(year) ? " Yes, it is!" : " No, it's not!"));
  }

  public static void main(String[] args) {
    int year = 2000;
    reportLeapYear(year);
  }
}
```

breakpoint here

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



↑ *returned value for the caller*

↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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

↑ *returned value for the caller*

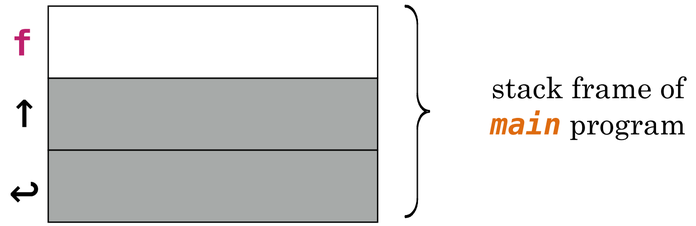
↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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



↑ *returned value for the caller*

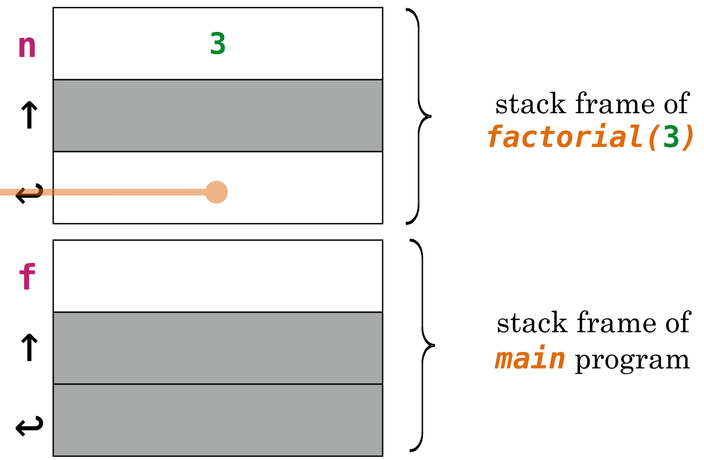
↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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

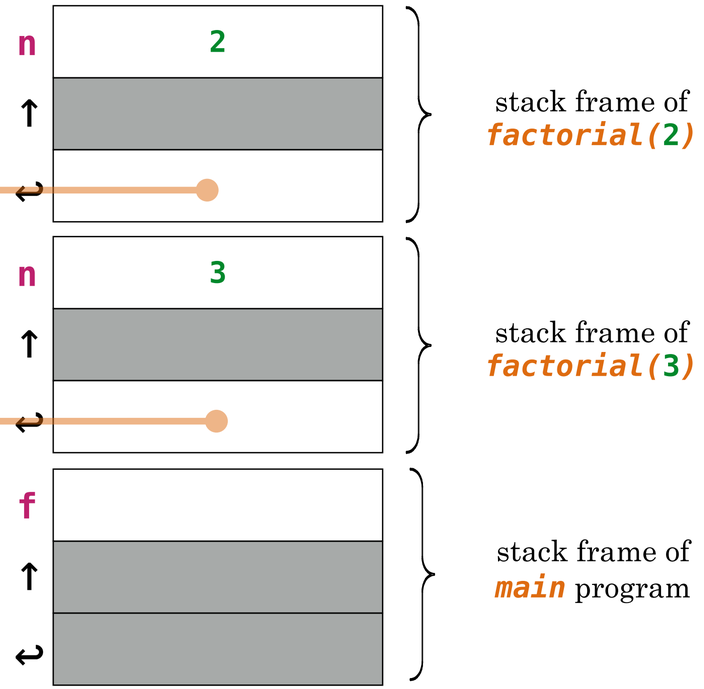


↑ *returned value for the caller*

↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```



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

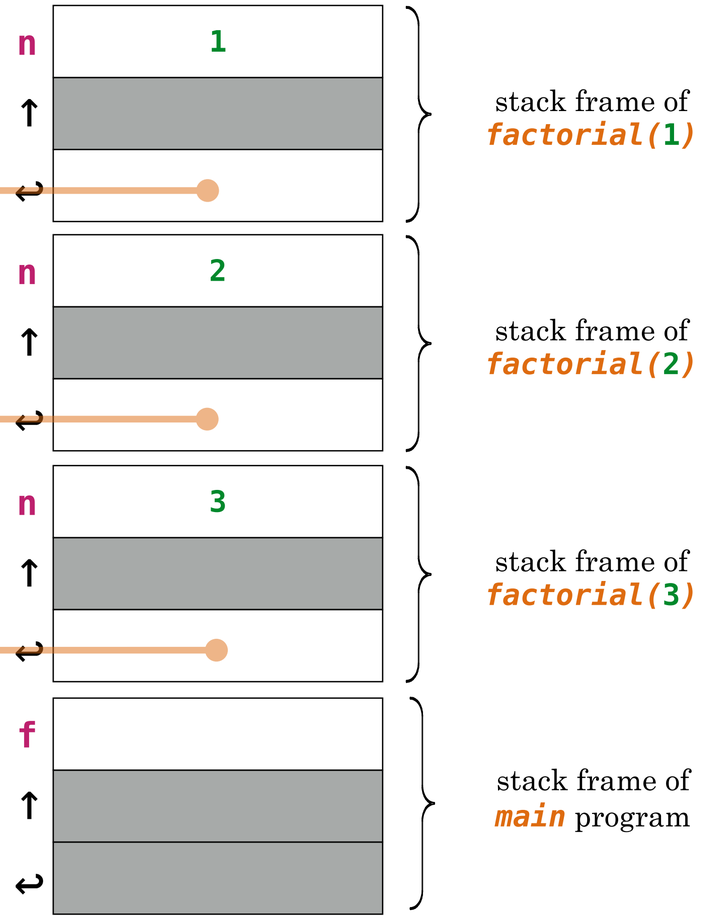
a stack frame contains all the local variables and parameters of the function being called

↑ *returned value for the caller*

↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```



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

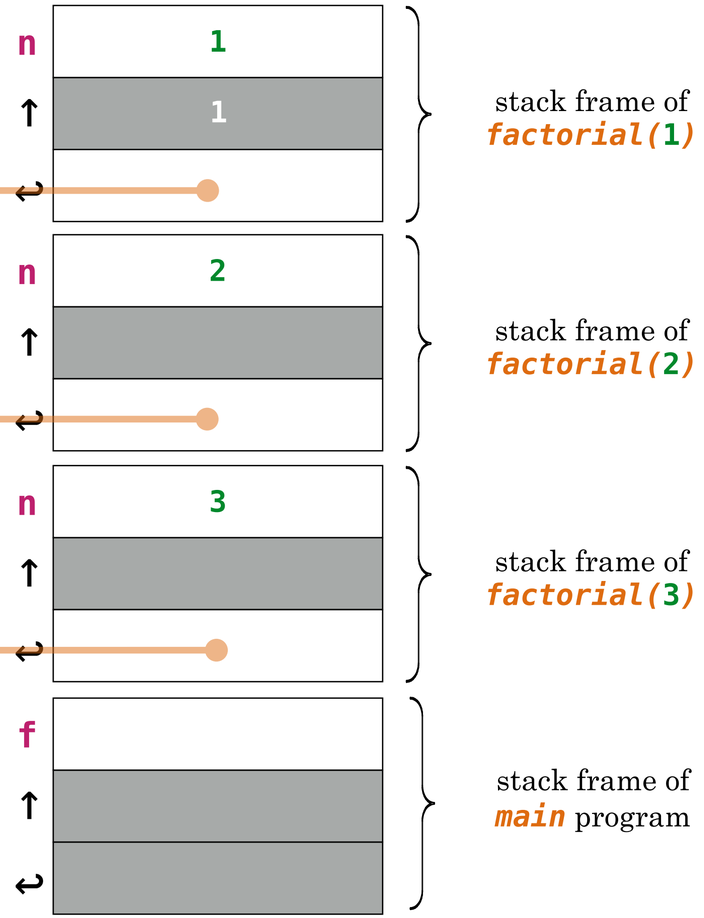
a stack frame contains all the local variables and parameters of the function being called

↑ *returned value for the caller*

↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```



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

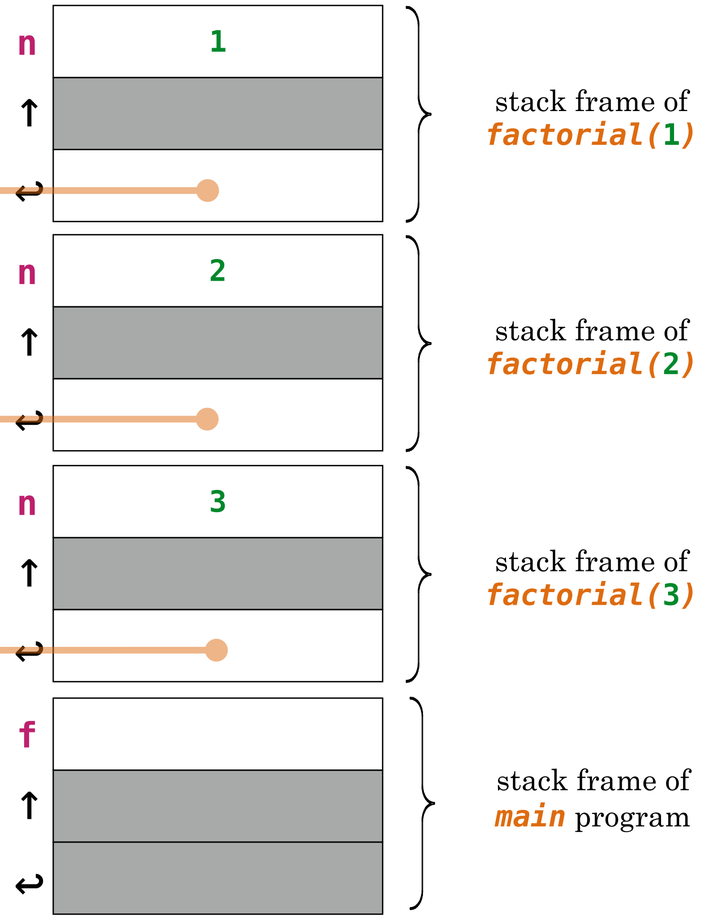
a stack frame contains all the local variables and parameters of the function being called

↑ *returned value for the caller*

↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```



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

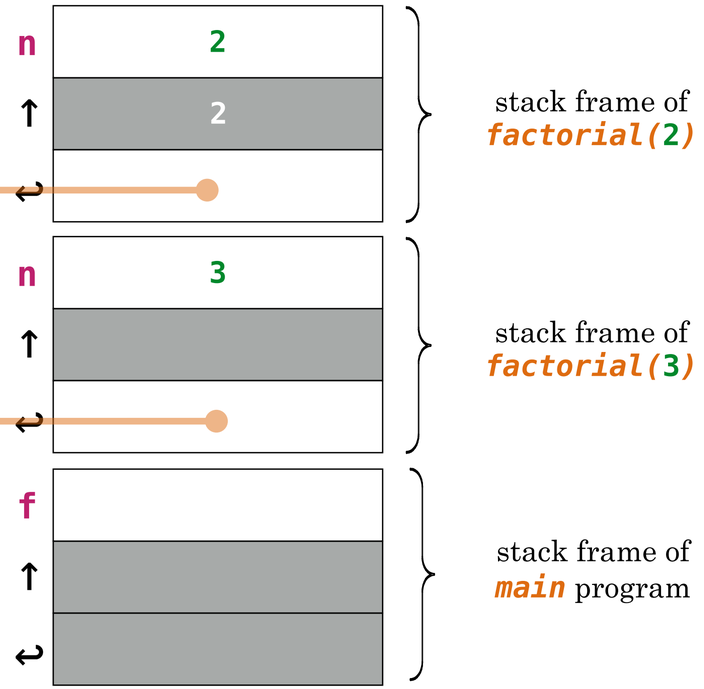
a stack frame contains all the local variables and parameters of the function being called

↑ *returned value for the caller*

↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```



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

↑ *returned value for the caller*

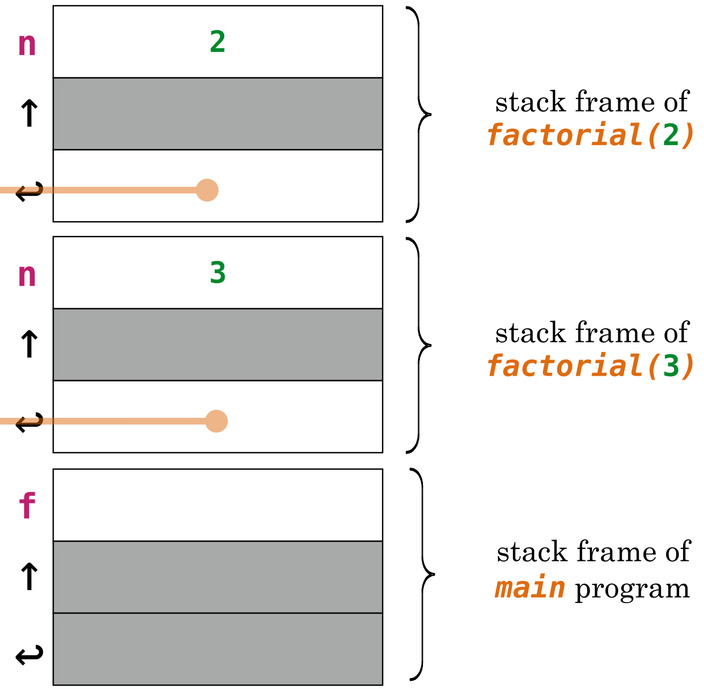
↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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



↑ *returned value for the caller*

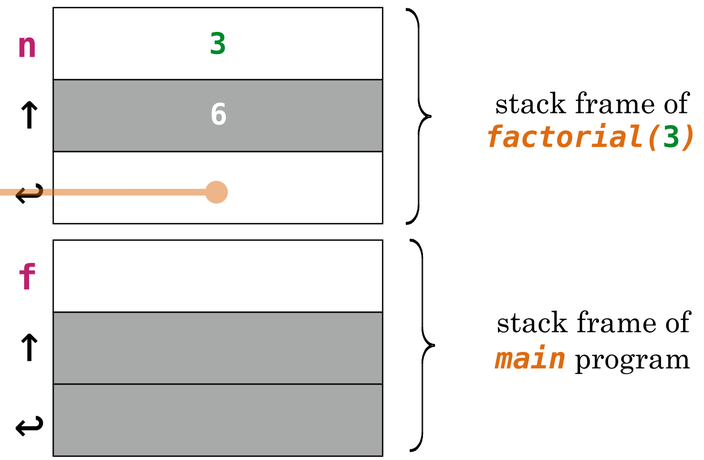
↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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



↑ *returned value for the caller*

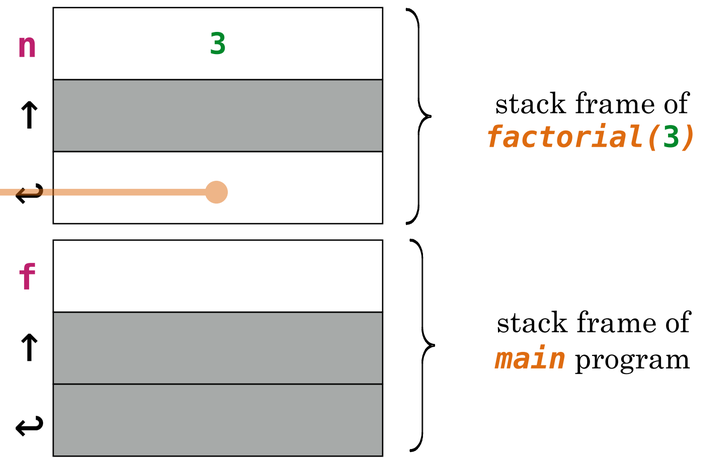
↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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



↑ *returned value for the caller*

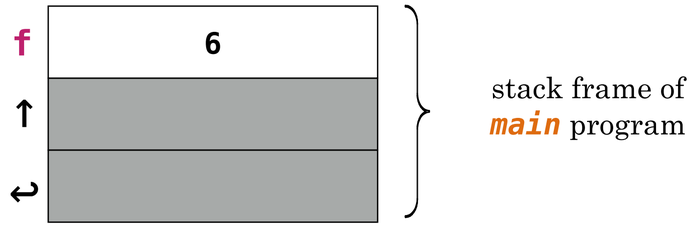
↩ *return address in the caller*

call stack - recursion

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)


f = factorial(3)
```

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



↑ *returned value for the caller*

↩ *return address in the caller*