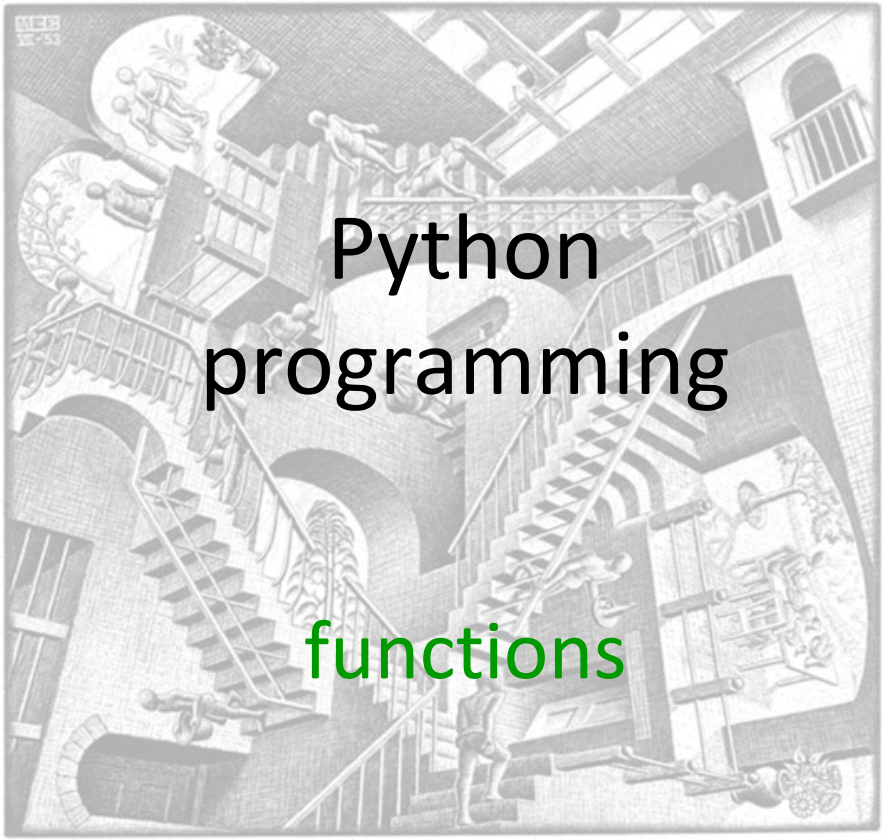
Ghent University
Faculty of Sciences



Python programming

functions

Prof. Dr. Peter Dawyndt

peter.dawyndt@ugent.be

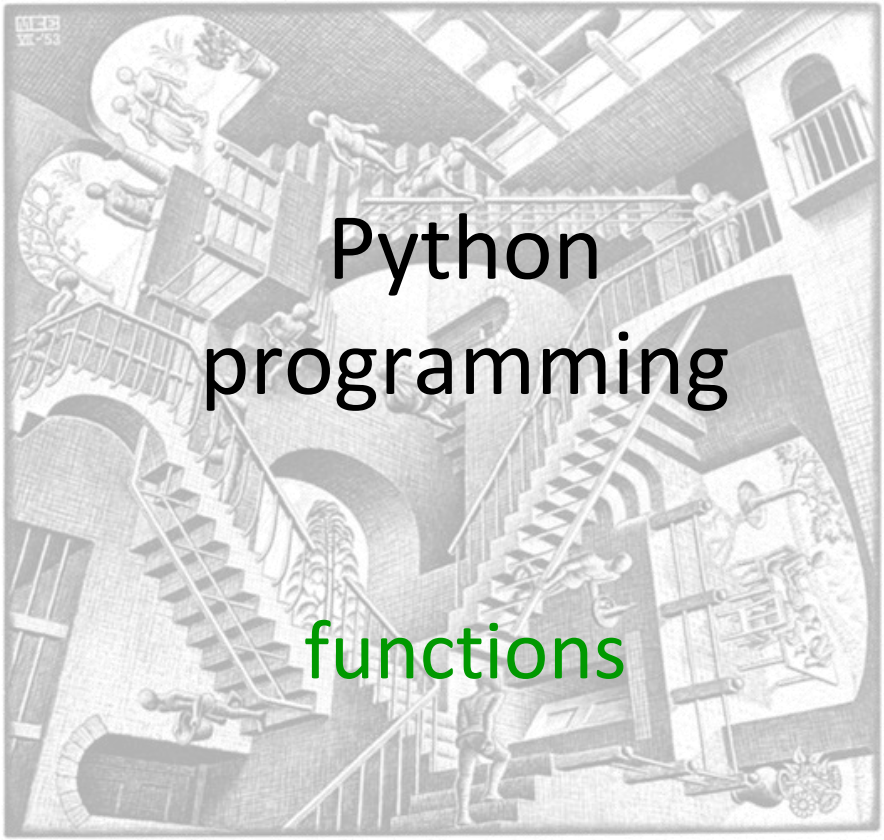
@dawyndt

UNIVERSITEIT GENT

“Civilization advances by extending the number of important operations which we can perform without thinking about them.”

— Alfred North Whitehead

Ghent University
Faculty of Sciences



Python programming

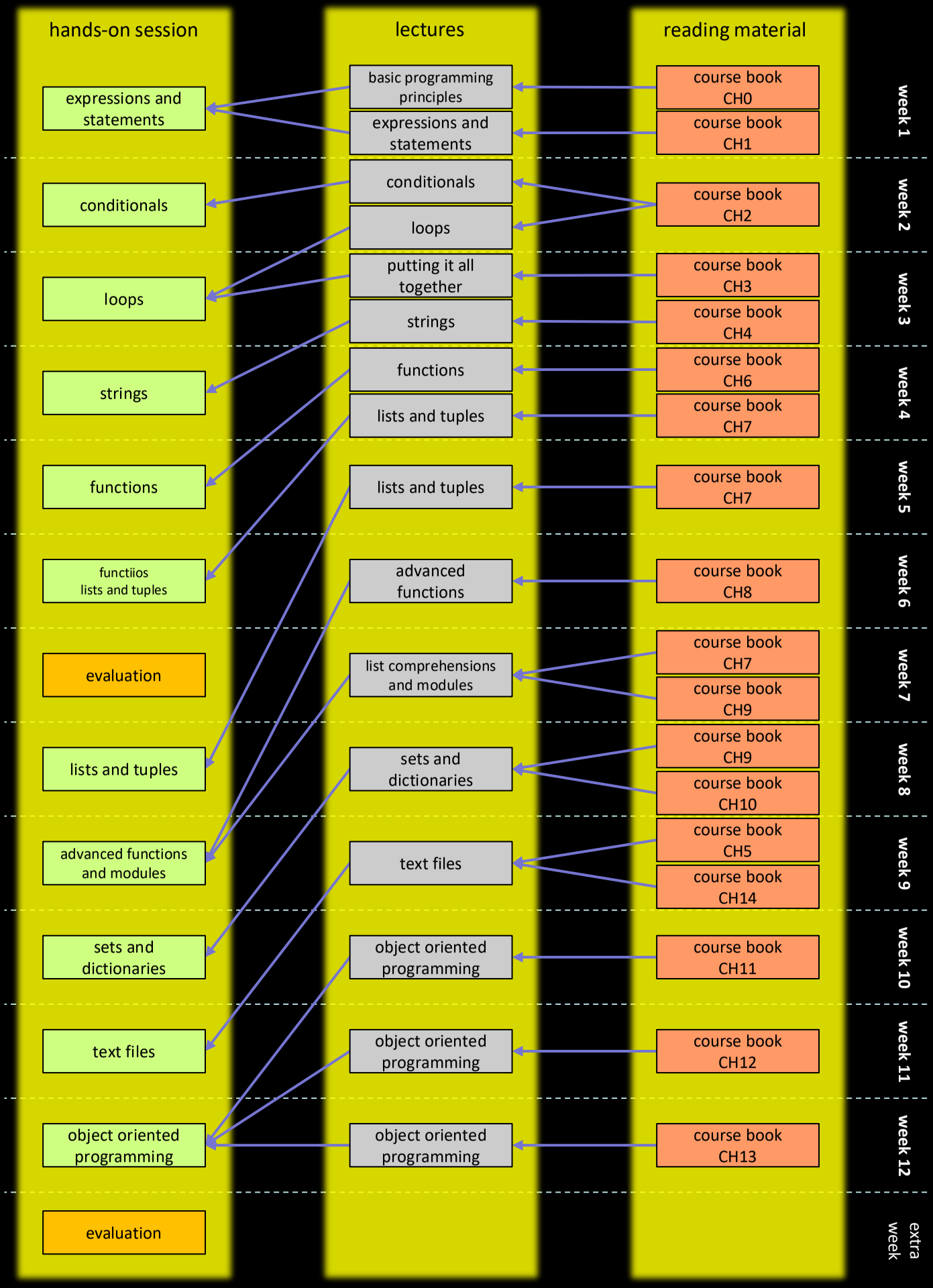
functions

Prof. Dr. Peter Dawyndt

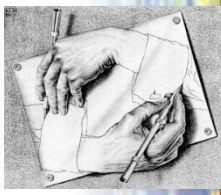
peter.dawyndt@ugent.be

@dawyndt

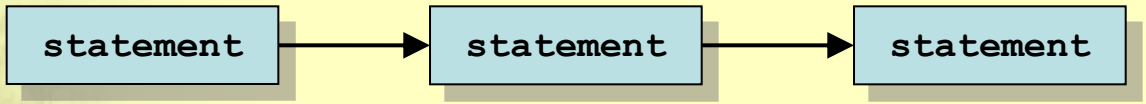
UNIVERSITEIT GENT

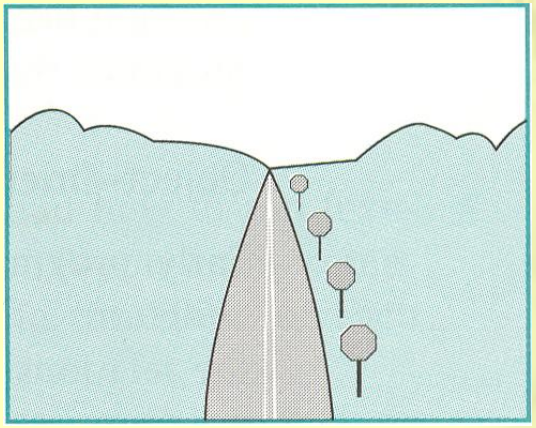

hands-on session
lectures
reading material
week 1
week 2
week 3
week 4
week 5
week 6
week 7
week 8
week 9
week 10
week 11
week 12
extra week
expressions and statements
conditionals
loops
strings
functions
functiios lists and tuples
evaluation
lists and tuples
advanced functions and modules
sets and dictionaries
text files
object oriented programming
evaluation
basic programming principles
expressions and statements
conditionals
loops
putting it all together
strings
functions
lists and tuples
lists and tuples
advanced functions
list comprehensions and modules
sets and dictionaries
text files
object oriented programming
object oriented programming
object oriented programming
course book CH0
course book CH1
course book CH2
course book CH3
course book CH4
course book CH6
course book CH7
course book CH7
course book CH8
course book CH7
course book CH9
course book CH9
course book CH10
course book CH5
course book CH14
course book CH11
course book CH12
course book CH13

Control structures

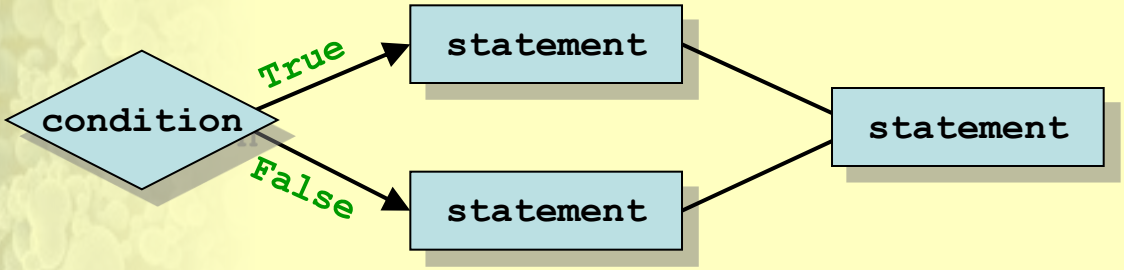


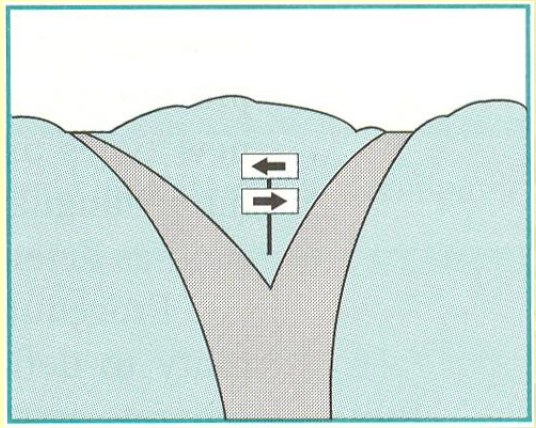
sequence



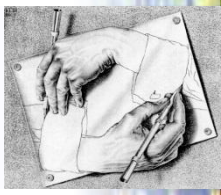


conditional statement (selection, jump)

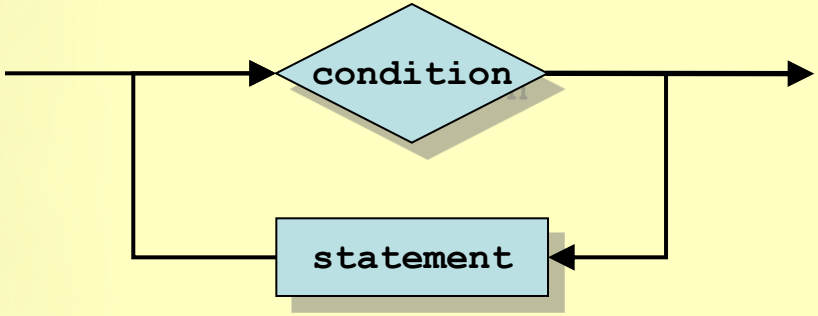


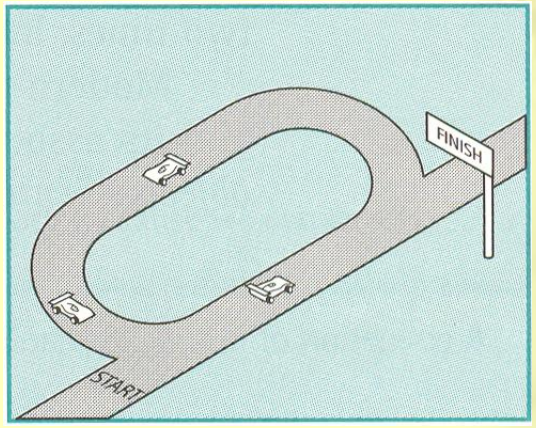


Control structures

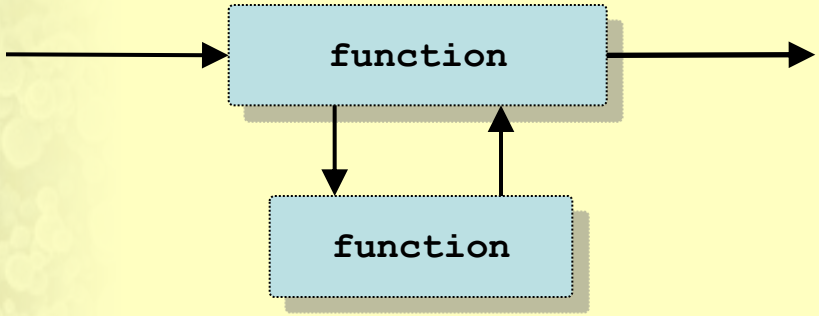


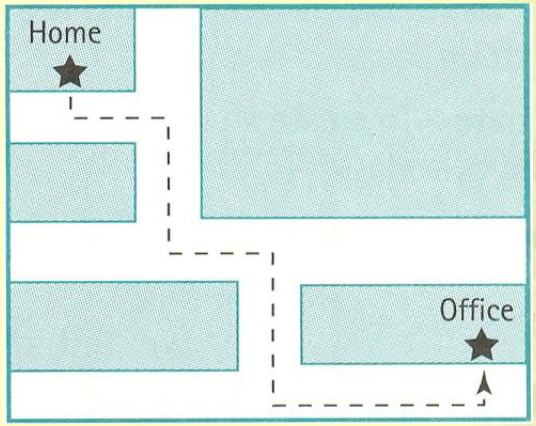
repetitive statetement (loop, iteration)





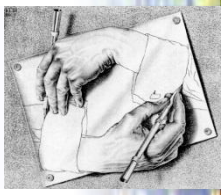
function (method)






I THINK YOU SHOULD BE MORE SPECIFIC HERE IN STEP TWO
THEN A MIRACLE OCCURS
S. Harris

Parameters and arguments

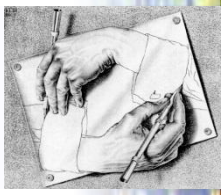


- **argument**: object (expression) passed when function is called
  - object is assigned to corresponding function parameter
- **parameter**: name used inside a function to refer to the value of an object that was passed as an argument

```
>>> abs(5)
5
>>> abs(-5)
5
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

UNIVERSITEIT GENT

Parameters and arguments

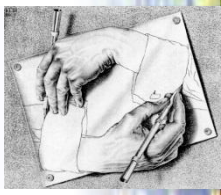


- **argument**: object (expression) passed when function is called
  - object is assigned to corresponding function parameter
- **parameter**: name used inside a function to refer to the value of an object that was passed as an argument

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5 ** 3, 512 - 9, 1024 ** 0)
503
>>> max(3 * 11, pow(5, 3), 512 - 9, pow(1024, 0))
503
```

UNIVERSITEIT GENT

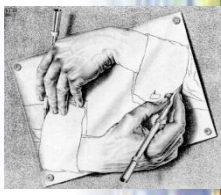
Design philosophy



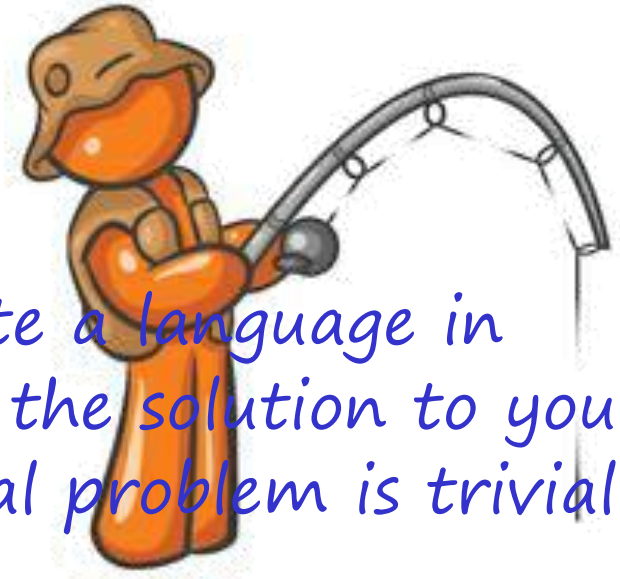
- programming languages should *not* try to include everything that anyone might ever want
- instead, they should make it easy for people to create what they need to solve specific problems
  - define functions to create higher-level operations (*abstraction*)

UNIVERSITEIT GENT

Design philosophy



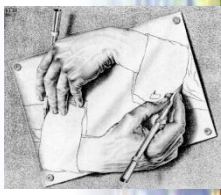
- programming languages should *not* try to include everything that anyone might ever want
- instead, they should make it easy for people to create what they need to solve specific problems
  - define functions to create higher-level operations (*abstraction*)



"Create a language in which the solution to your original problem is trivial."

UNIVERSITEIT GENT

Defining functions



- keyword **`def`** defines a new function
  - names of parameters are enclosed in parentheses
    - function definition: zero or more parameters
  - data type of parameters is not fixed

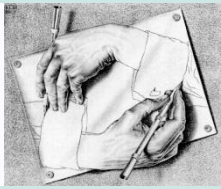
```
>>> def double(x):
...     return 2 * x
...
>>> double(12)
24
>>> double('lava')
'lavalava'
>>> double([1, 2, 3])
[1, 2, 3, 1, 2, 3]
```

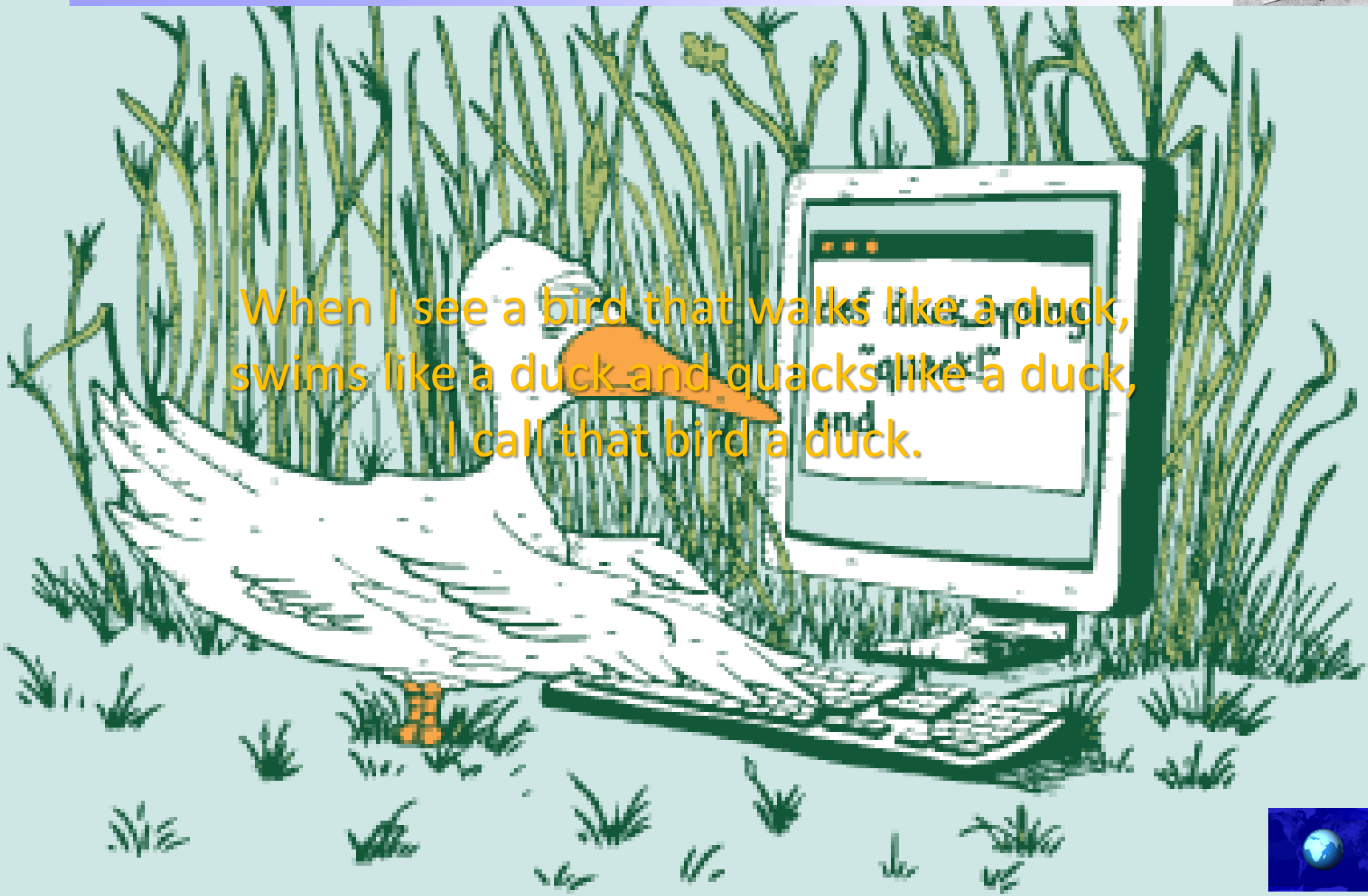
syntax

```
def name([parameter, …]):
    statements
```

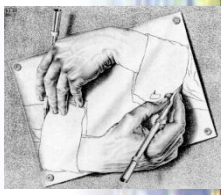
UNIVERSITEIT GENT

Duck typing



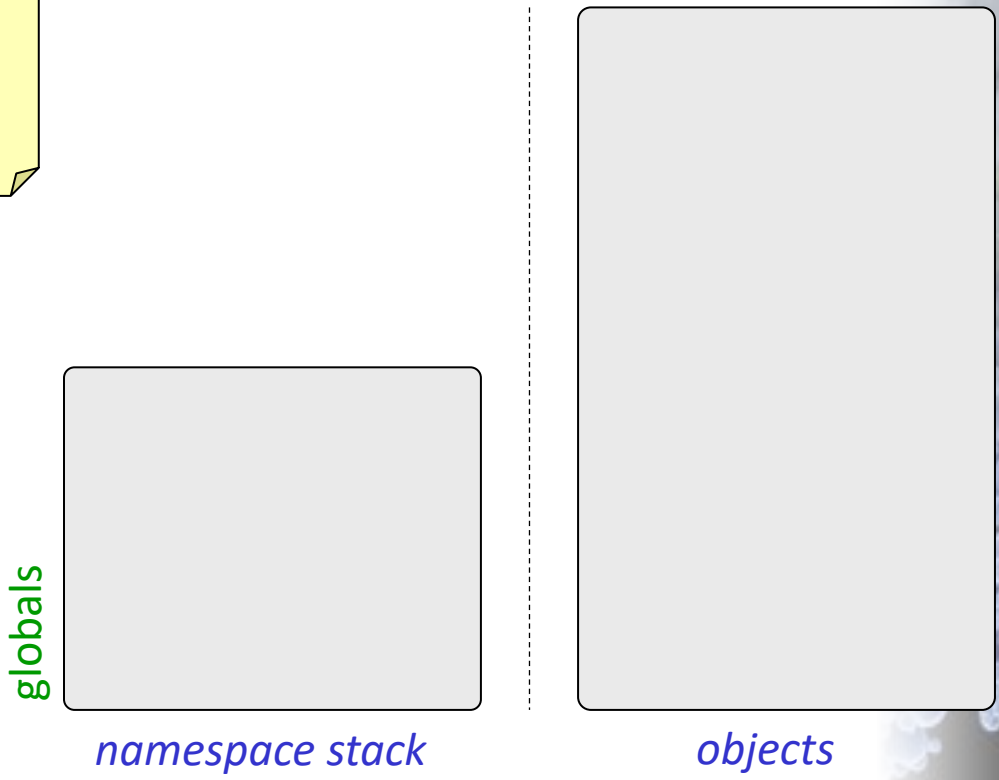

When I see a bird that walks like a duck,
swims like a duck and quacks like a duck,
I call that bird a duck.

Passing arguments



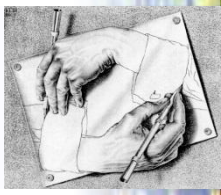
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

type = 'sandstone'
result = classify(stone)
```



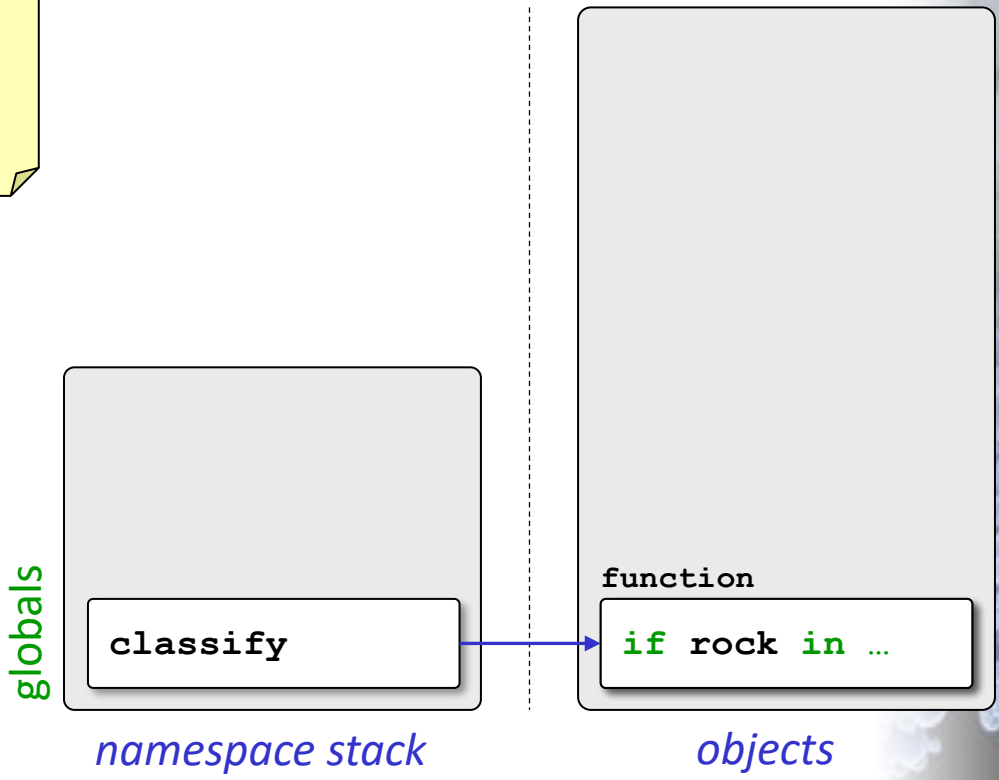
UNIVERSITEIT GENT

Passing arguments



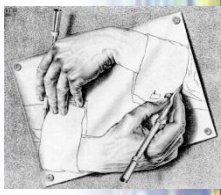
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



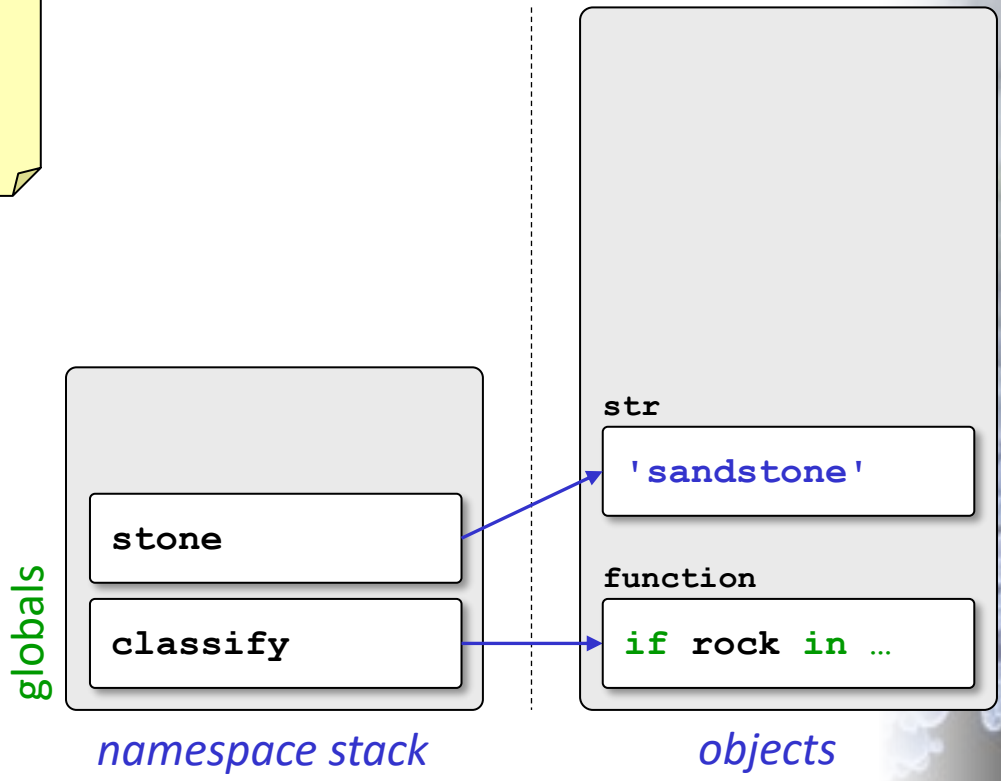
UNIVERSITEIT GENT

Passing arguments



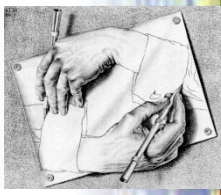
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



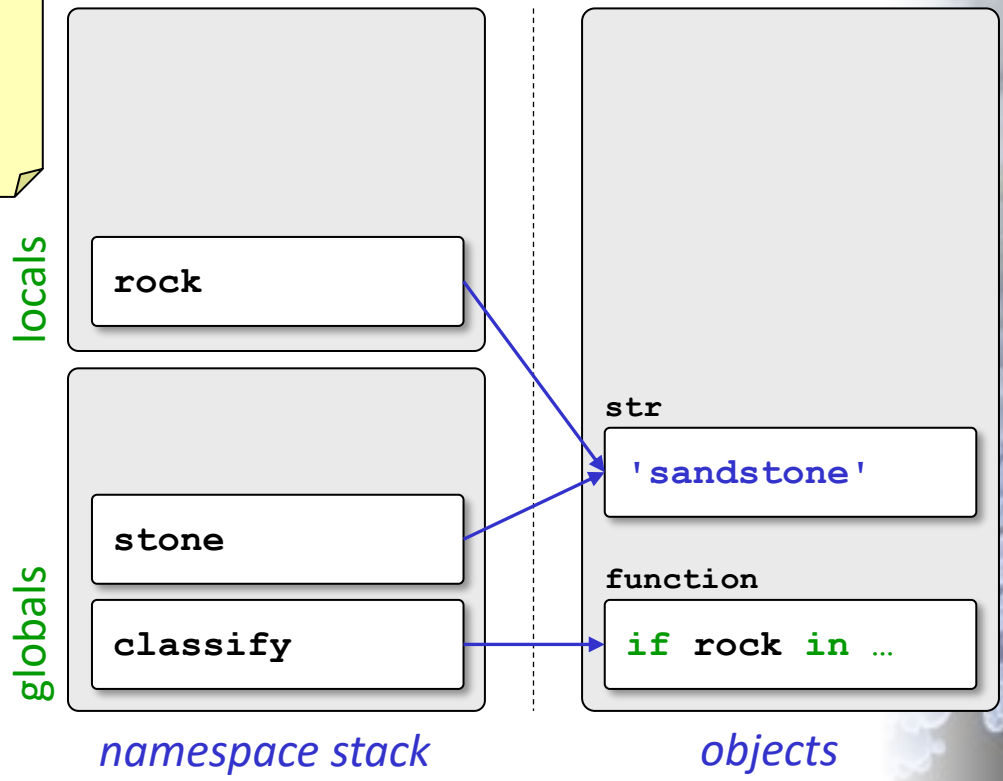
UNIVERSITEIT GENT

Passing arguments



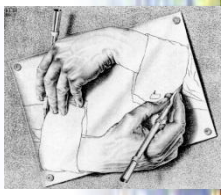
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



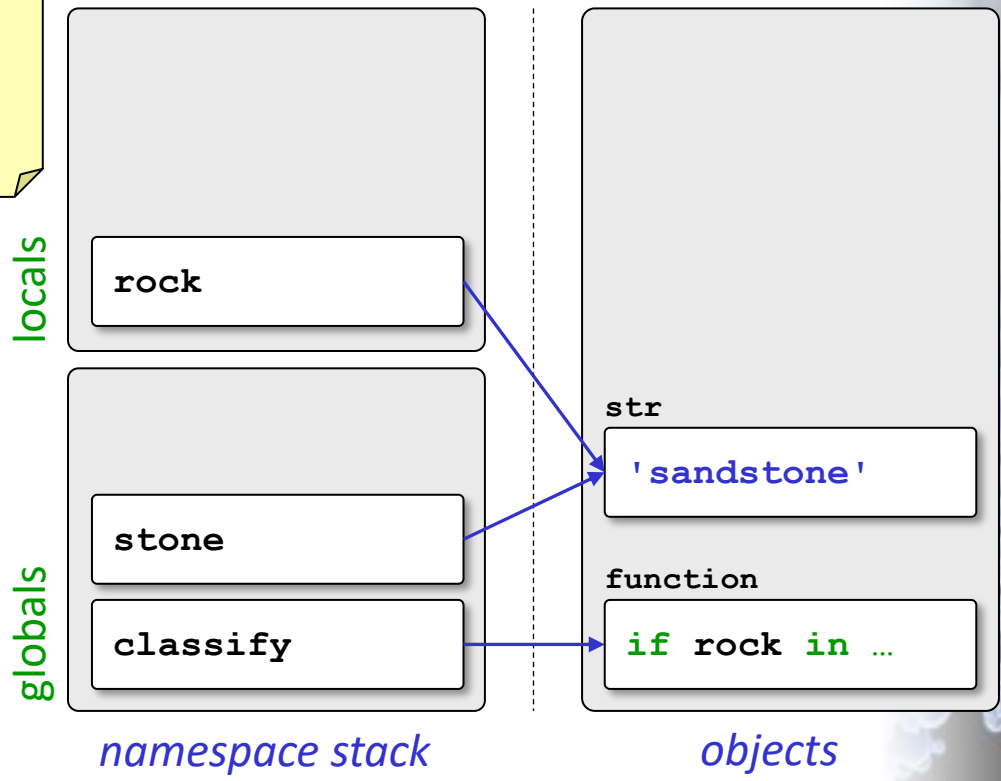
UNIVERSITEIT GENT

Passing arguments



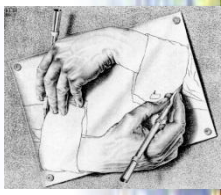
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



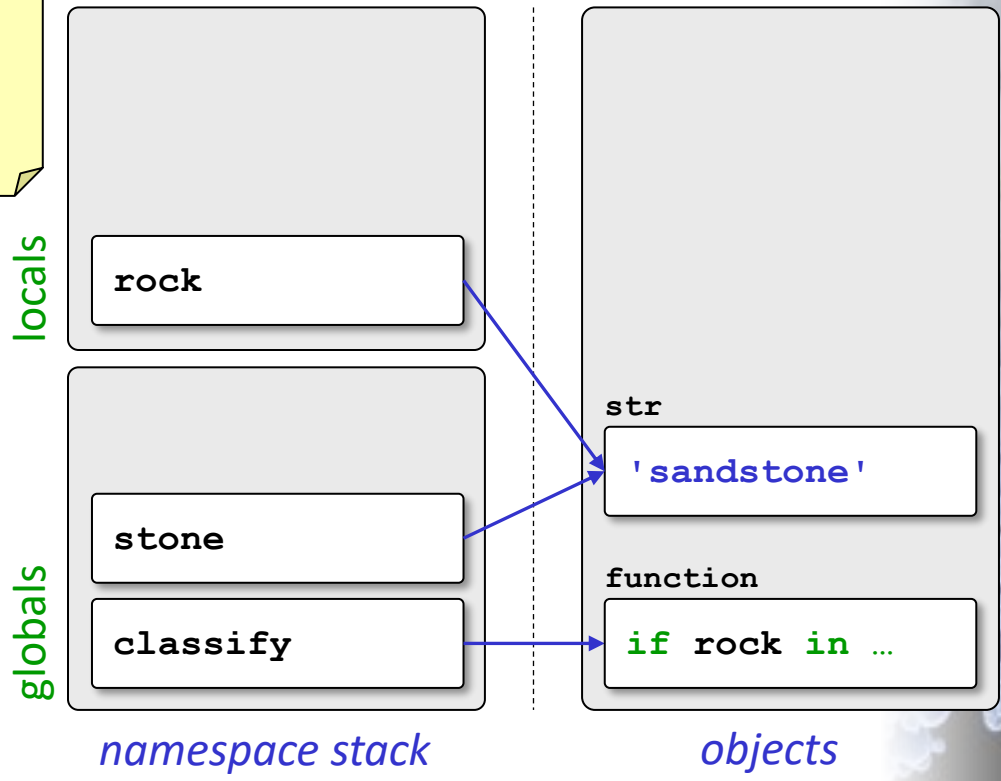
UNIVERSITEIT GENT

Passing arguments



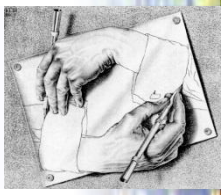
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



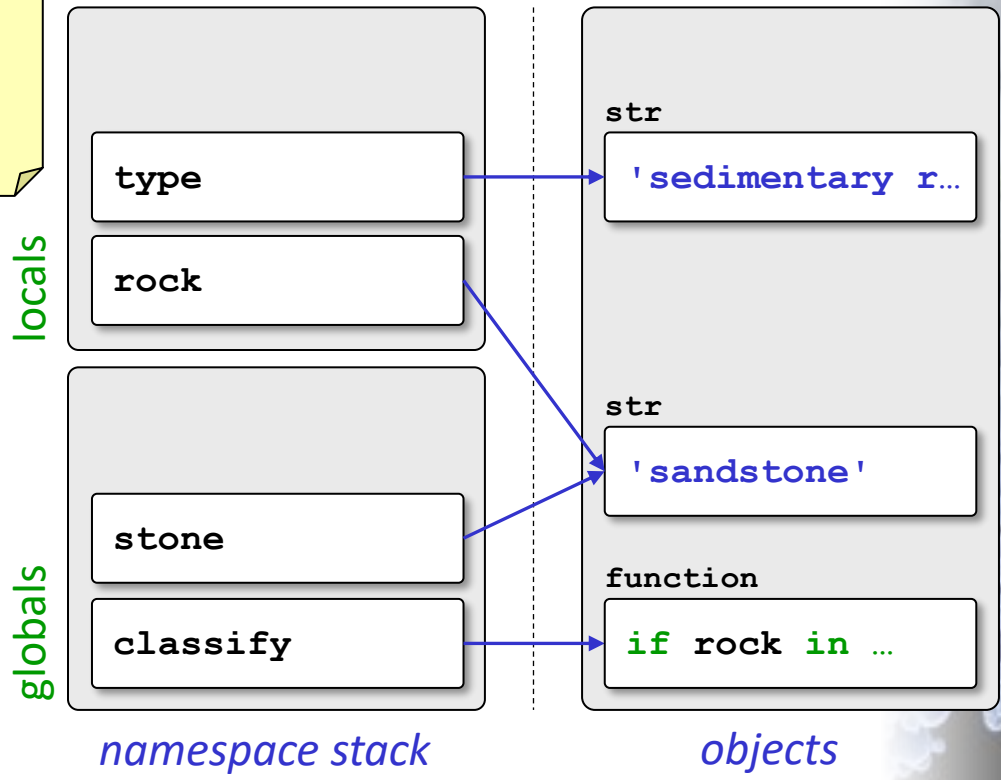
UNIVERSITEIT GENT

Passing arguments



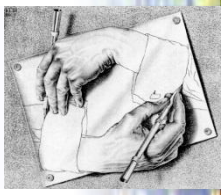
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



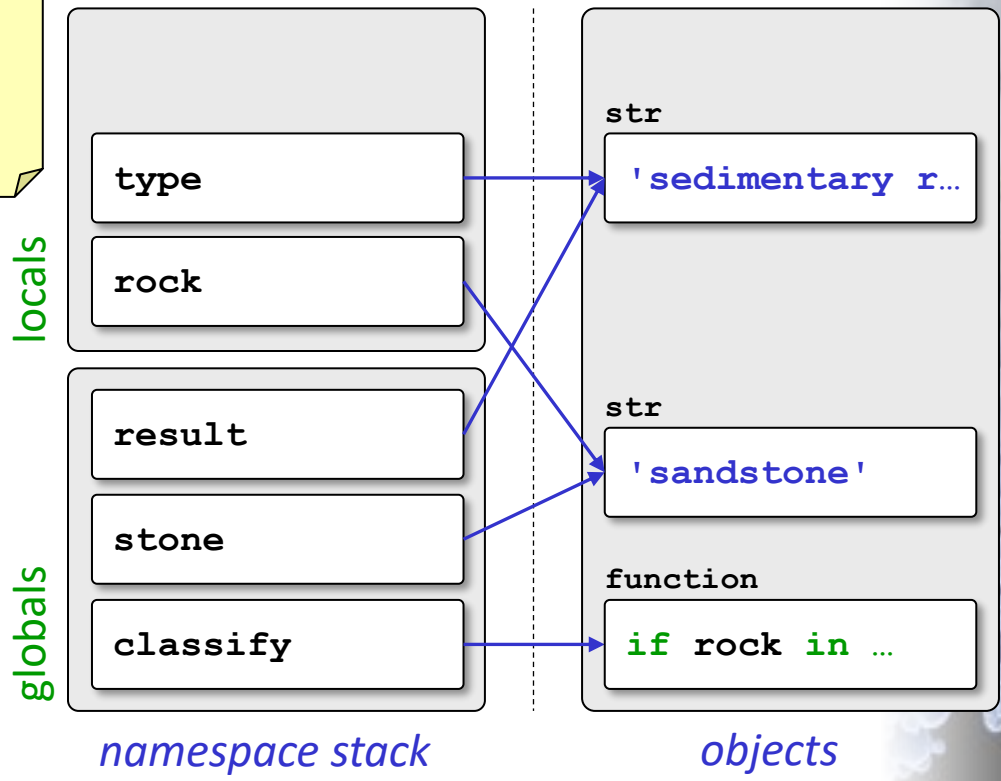
UNIVERSITEIT GENT

Returning objects



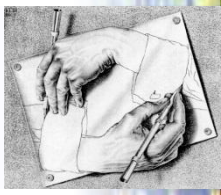
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



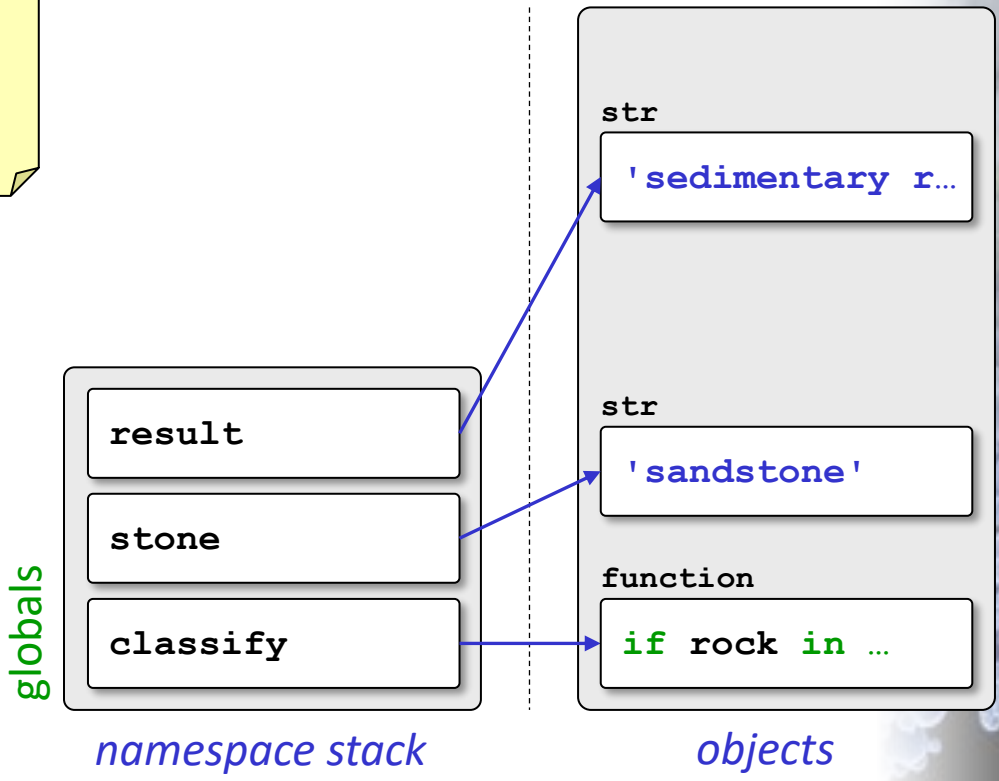
UNIVERSITEIT GENT

Returning objects



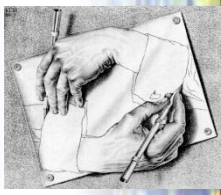
rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

stone = 'sandstone'
result = classify(stone)
```



UNIVERSITEIT GENT

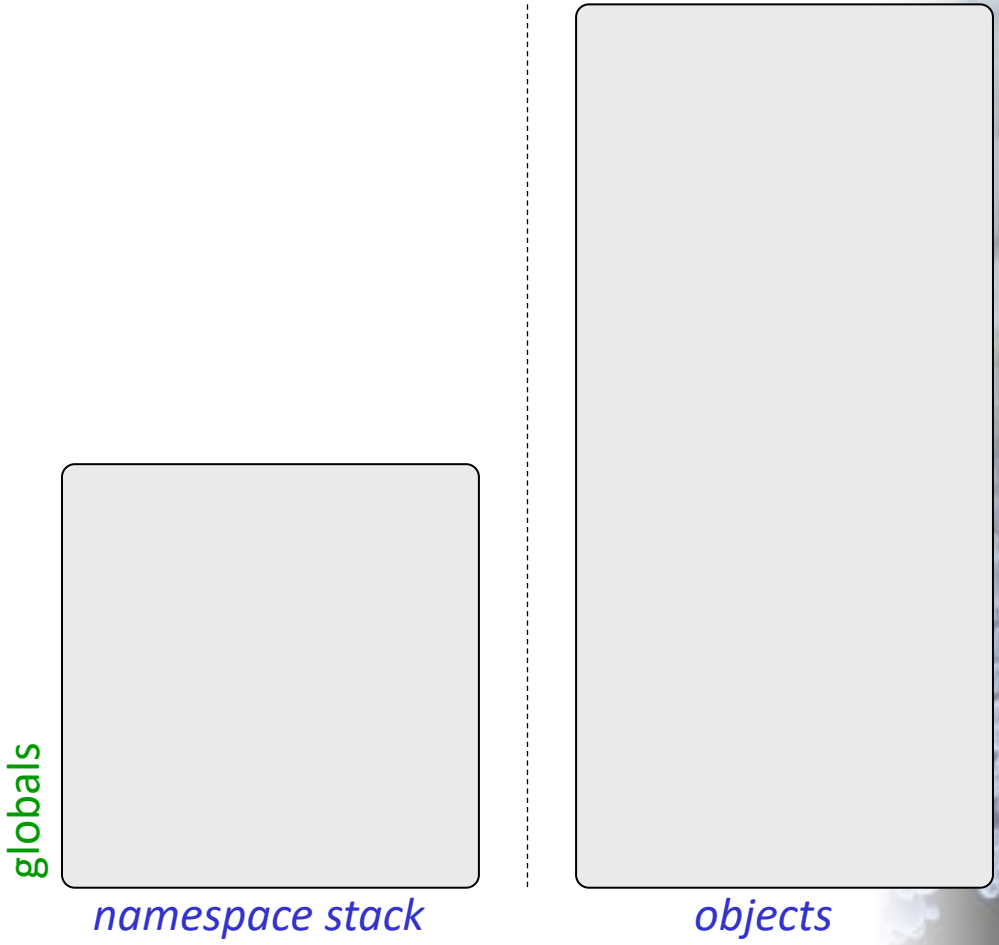
Calling functions



double.py

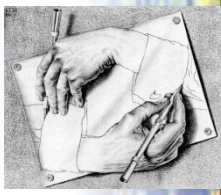
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

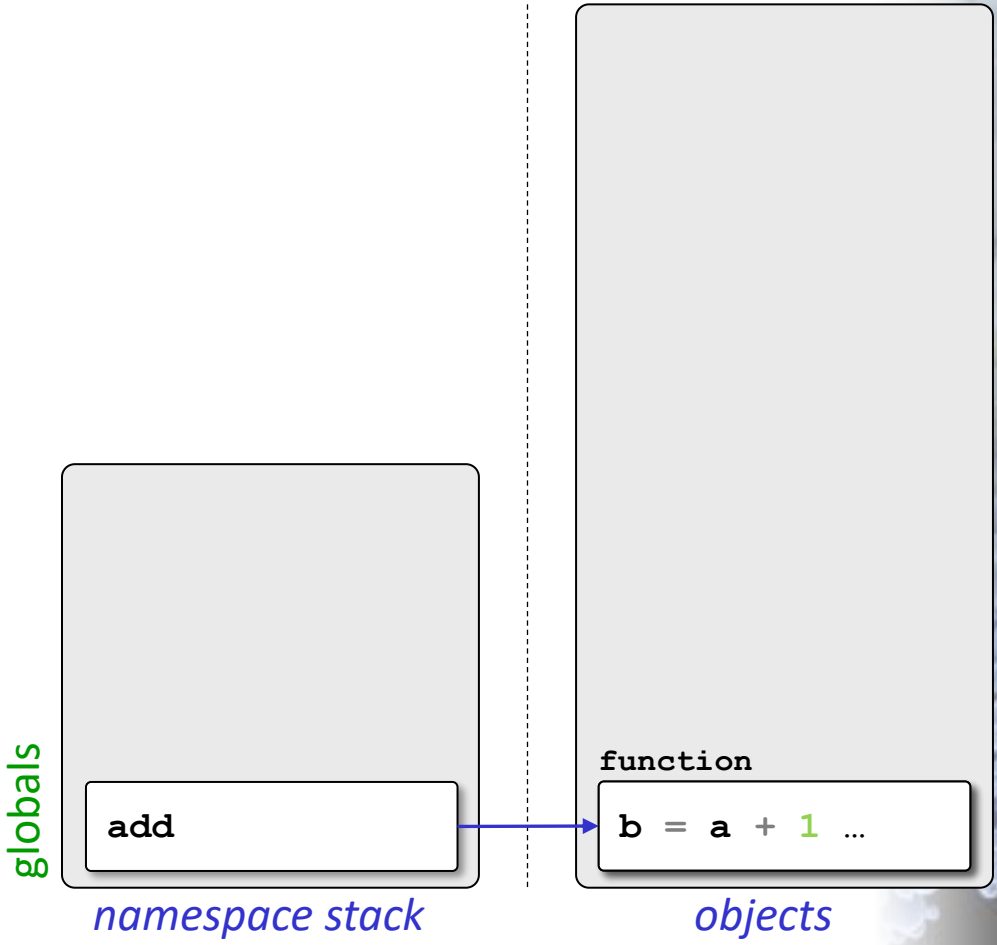
Calling functions



double.py

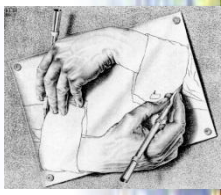
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

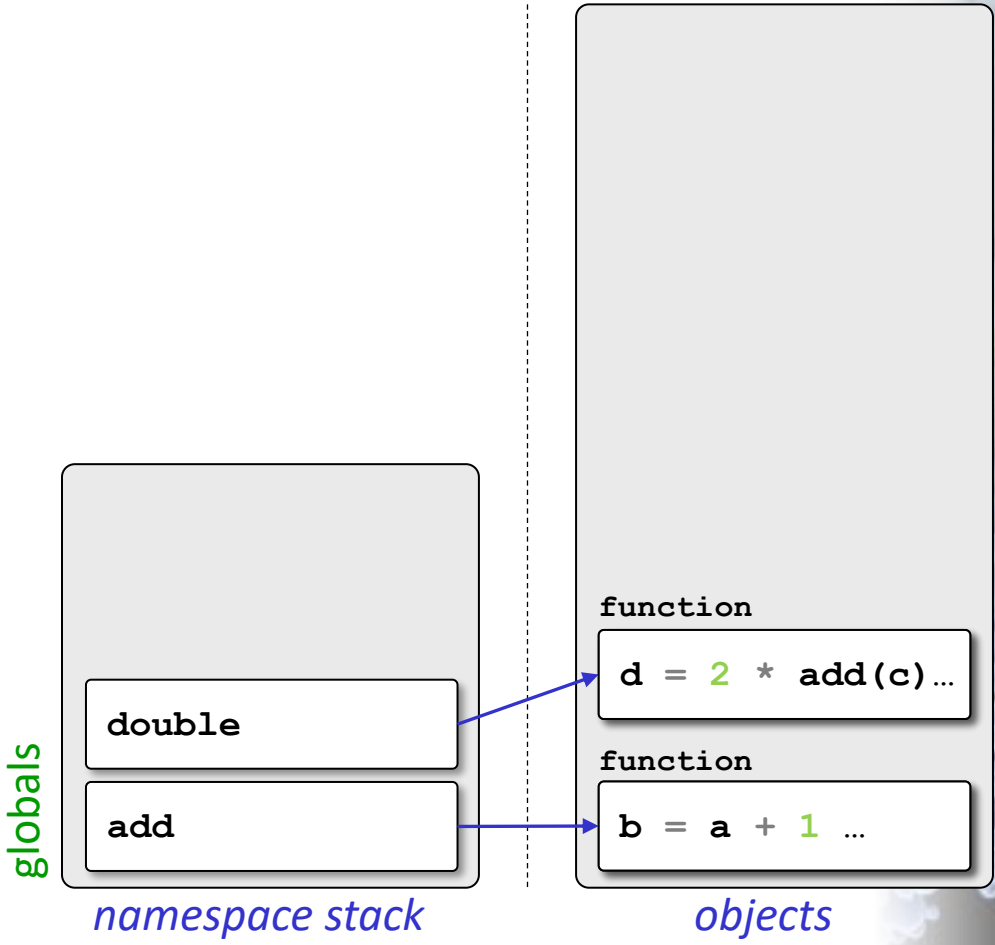
Calling functions



double.py

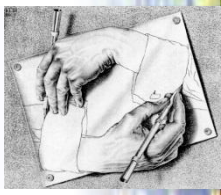
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

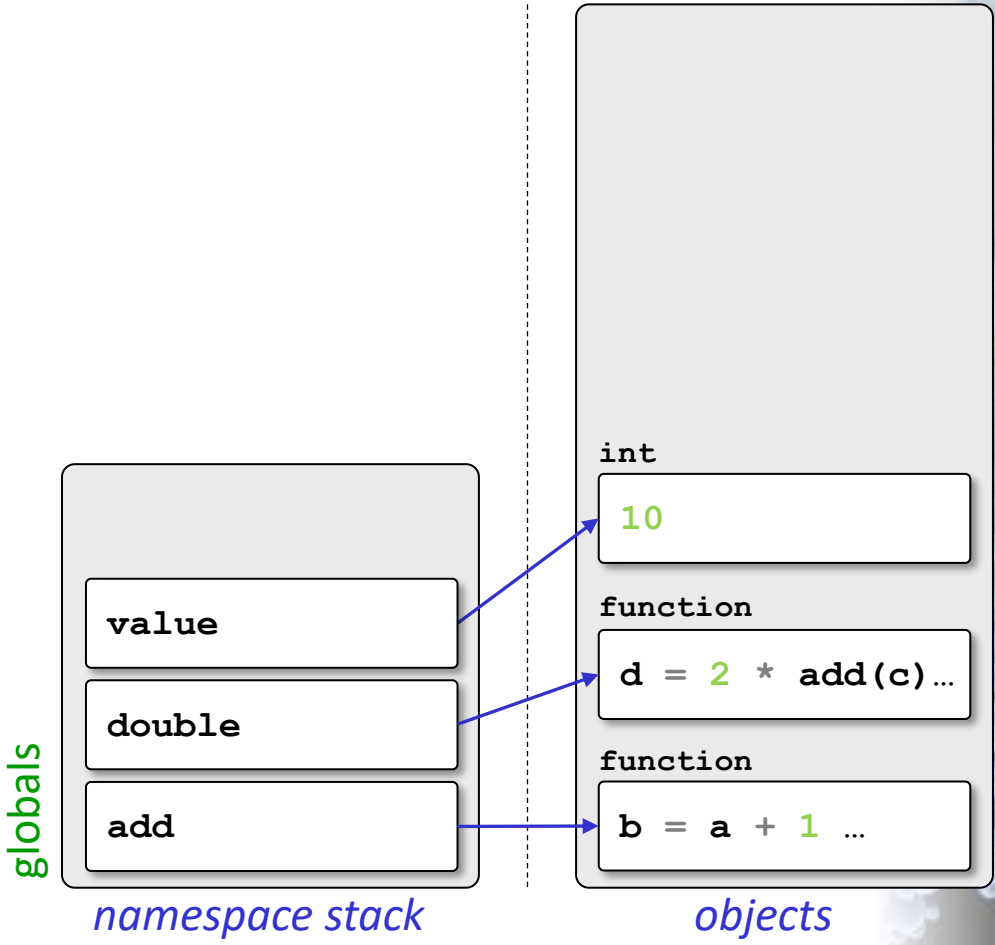
Calling functions



double.py

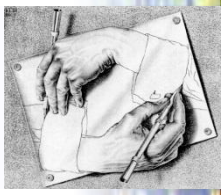
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

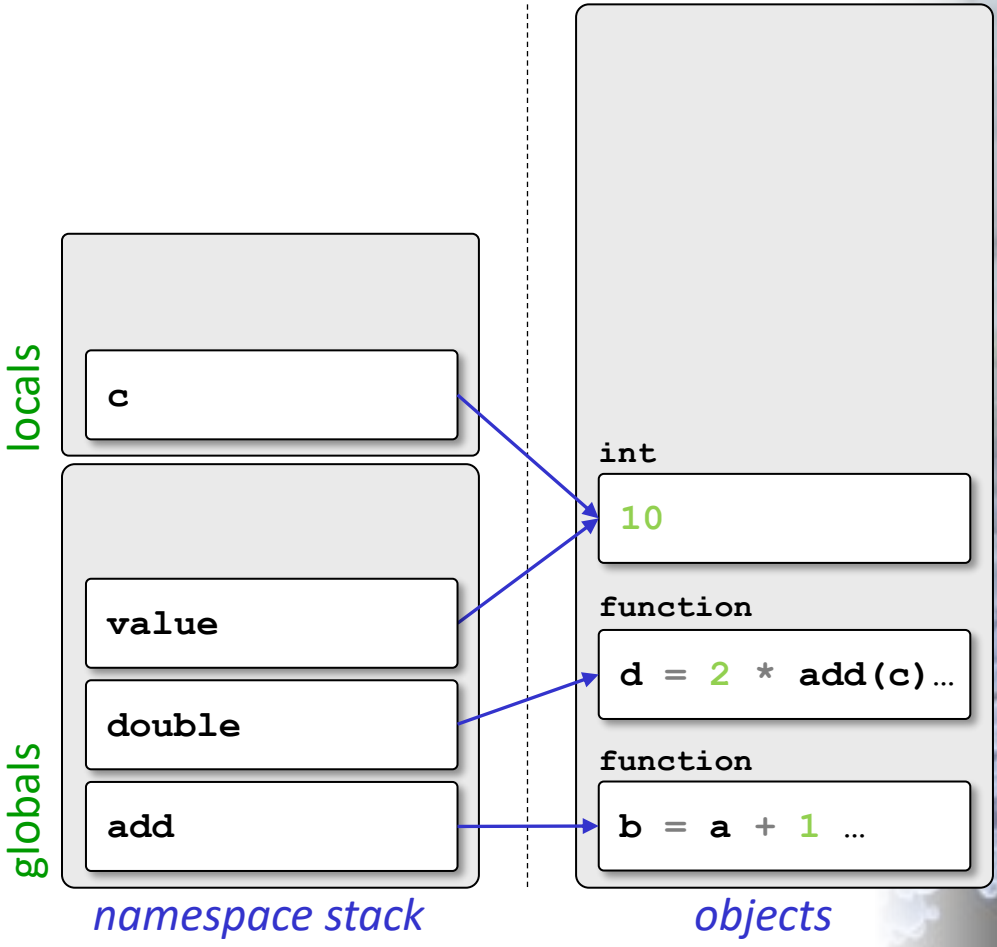
Calling functions



double.py

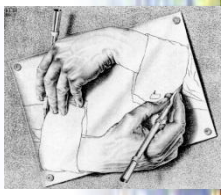
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

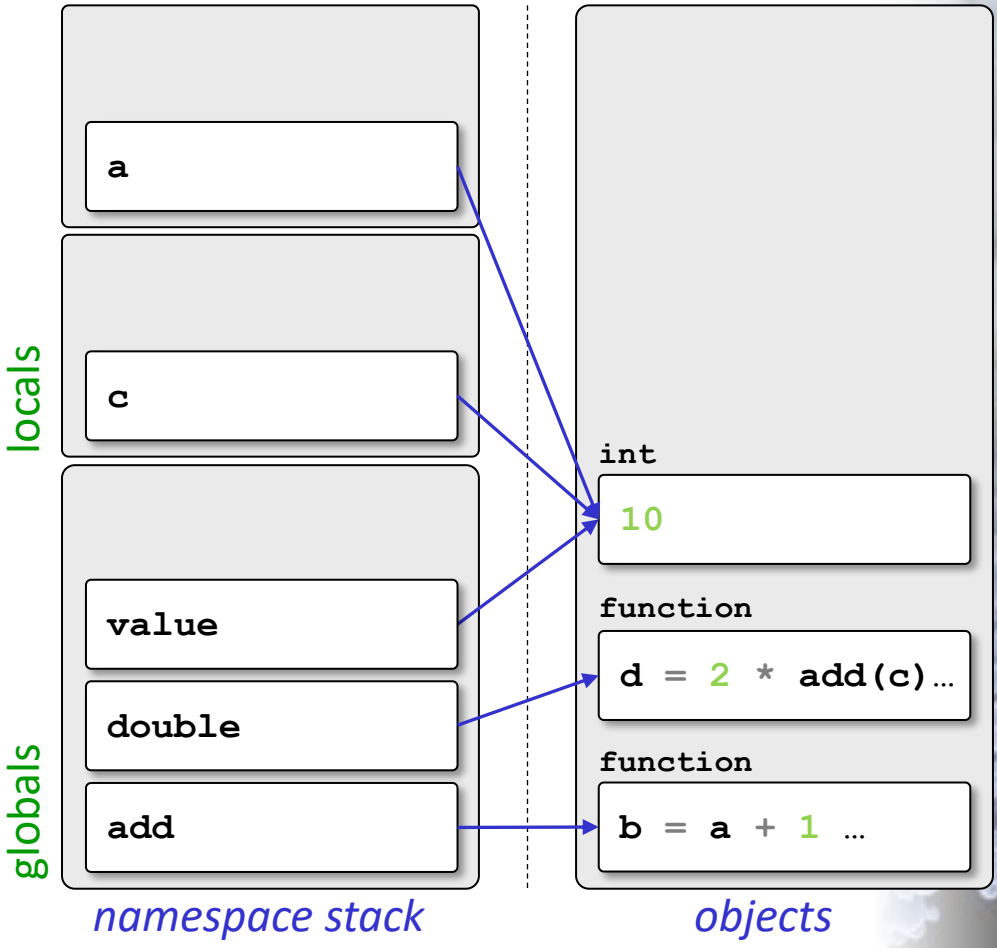
Calling functions



double.py

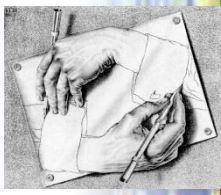
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

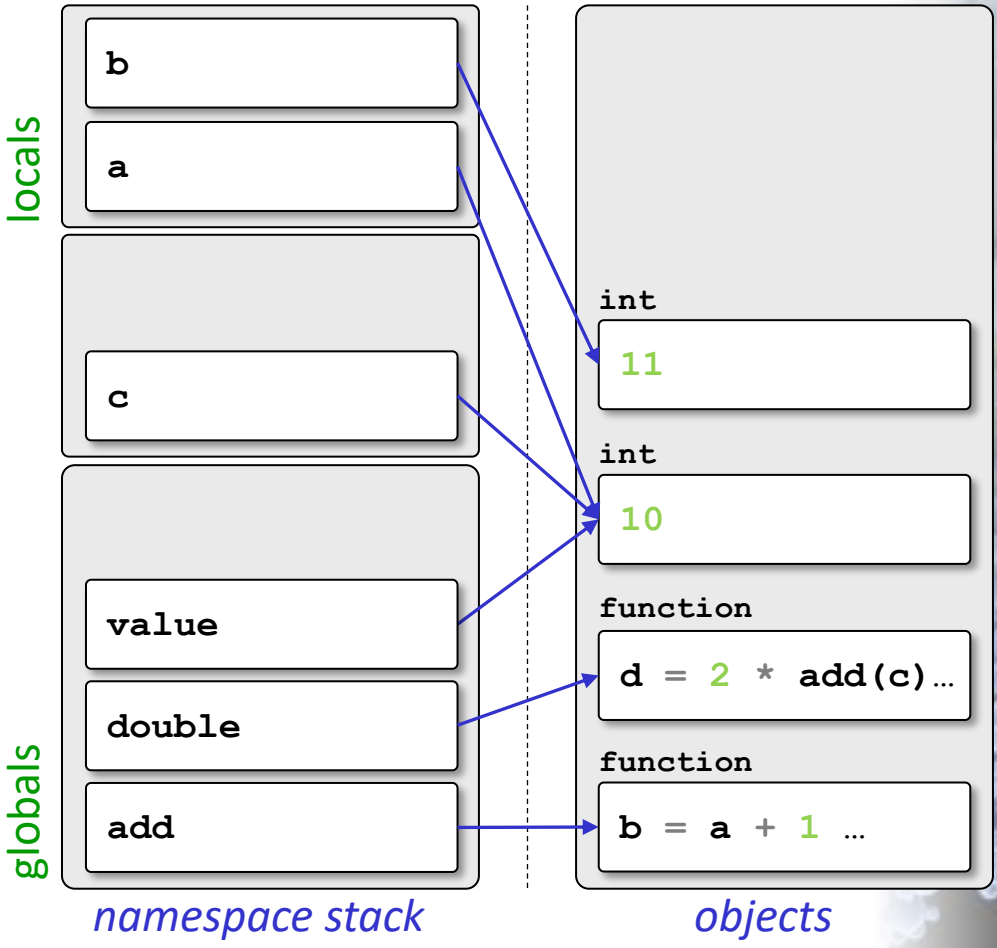
Calling functions



double.py

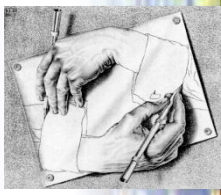
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

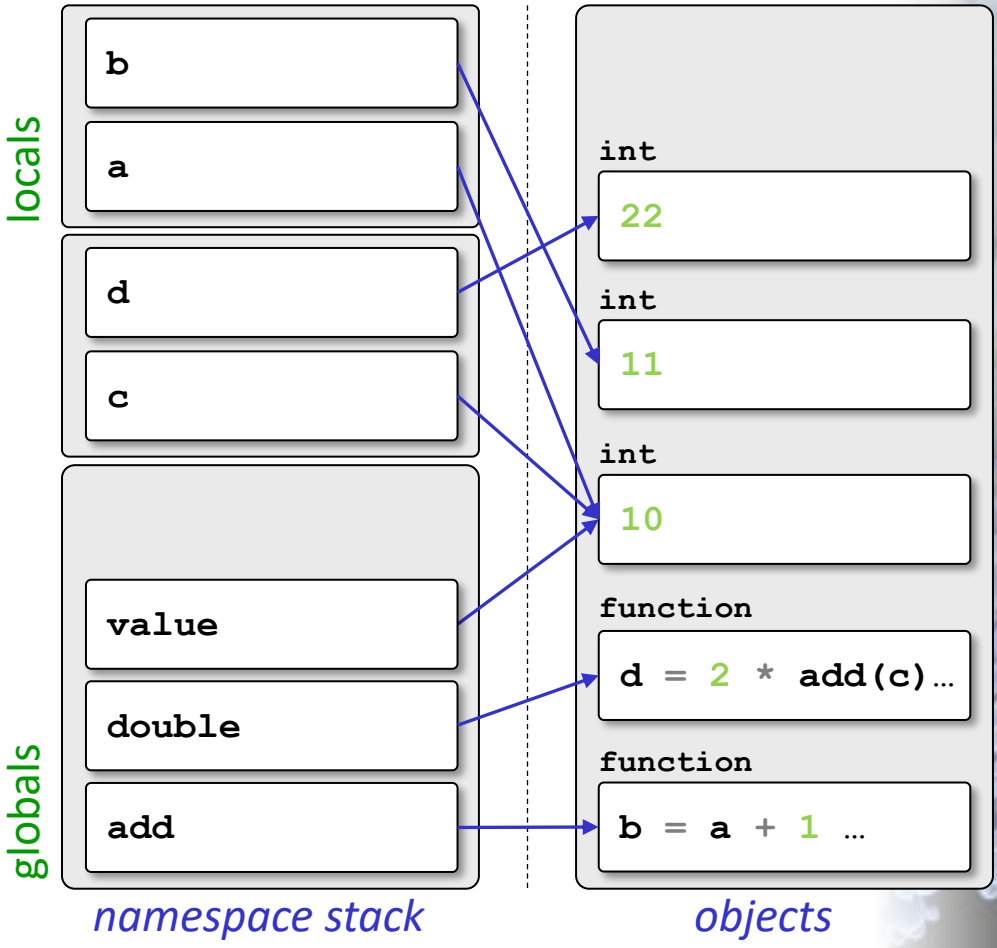
Calling functions



double.py

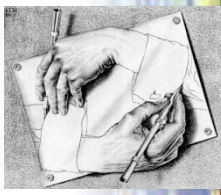
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

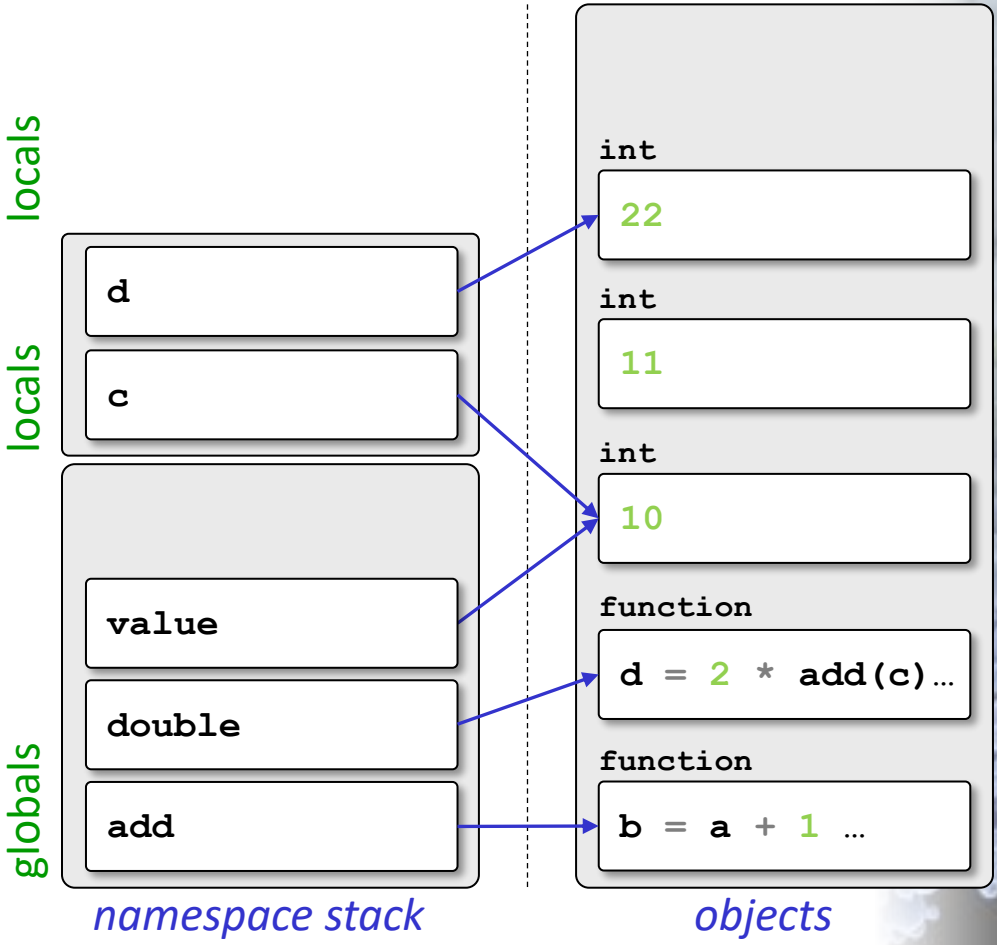
Calling functions



double.py

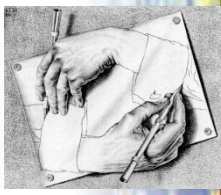
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

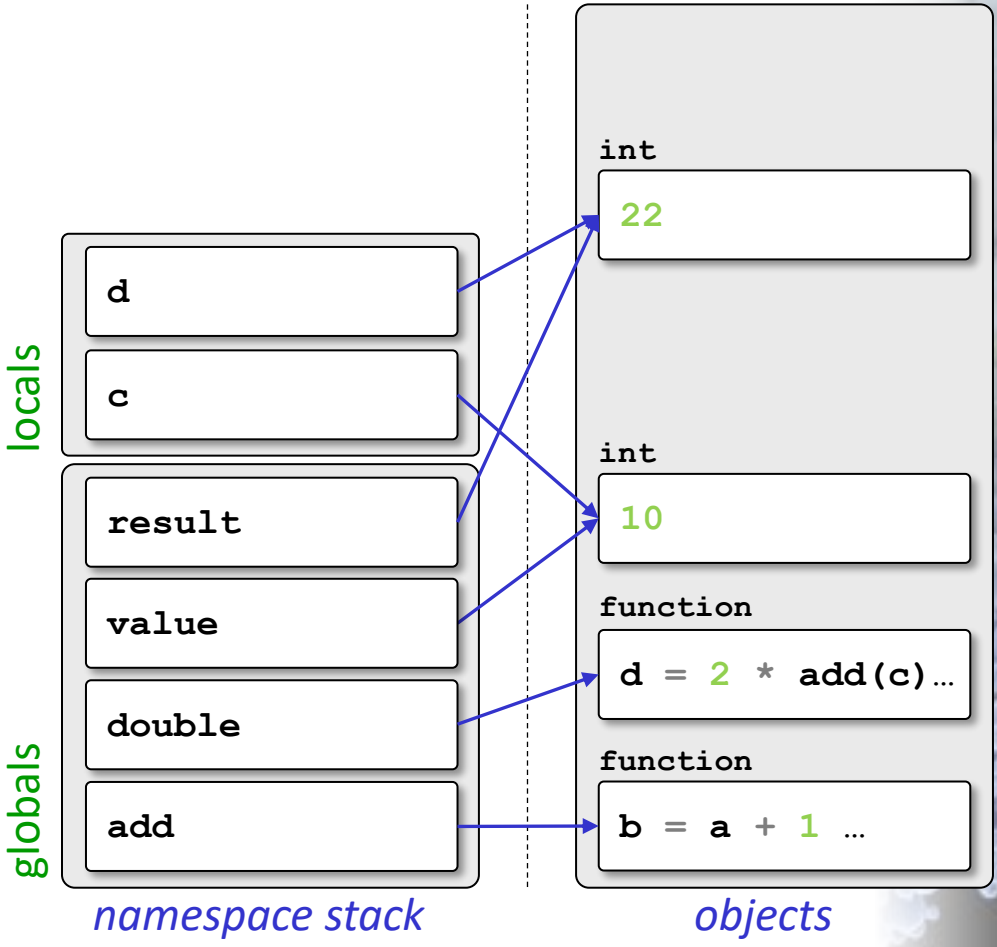
Calling functions



double.py

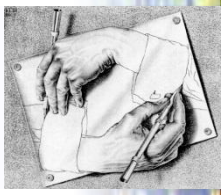
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

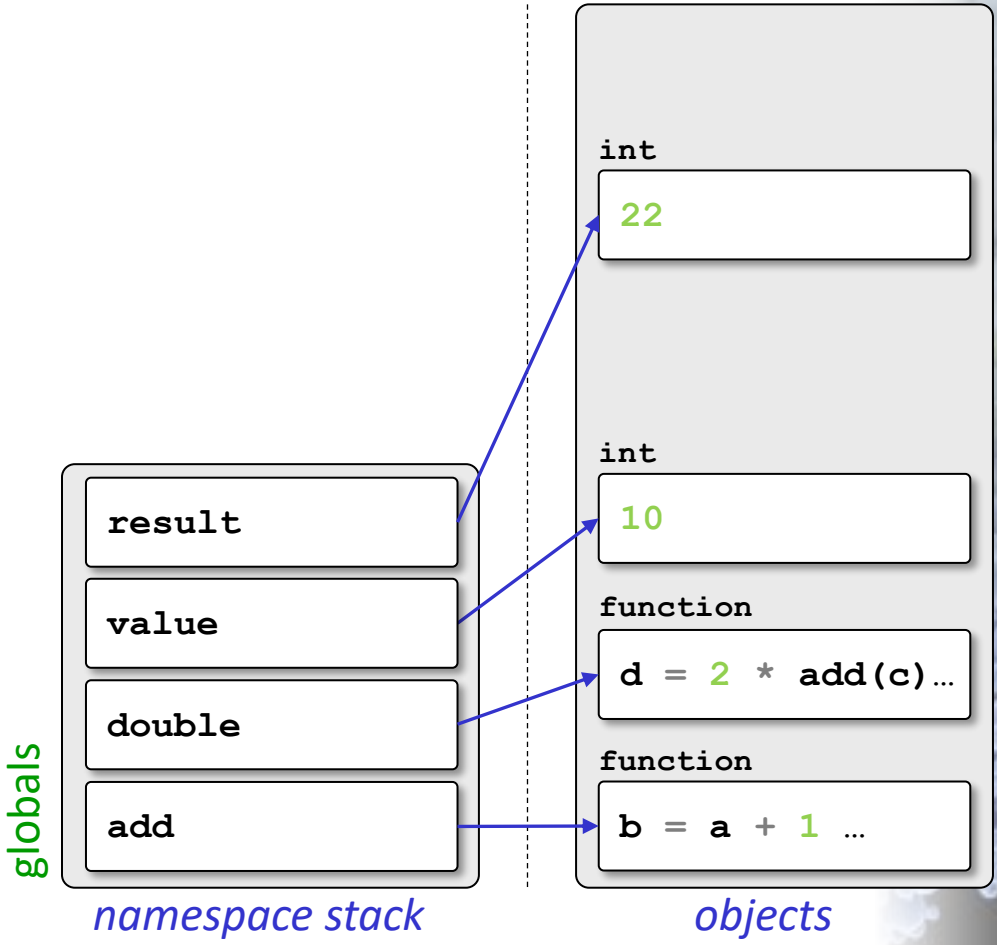
Calling functions



double.py

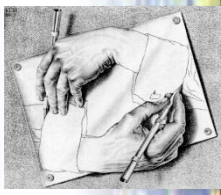
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

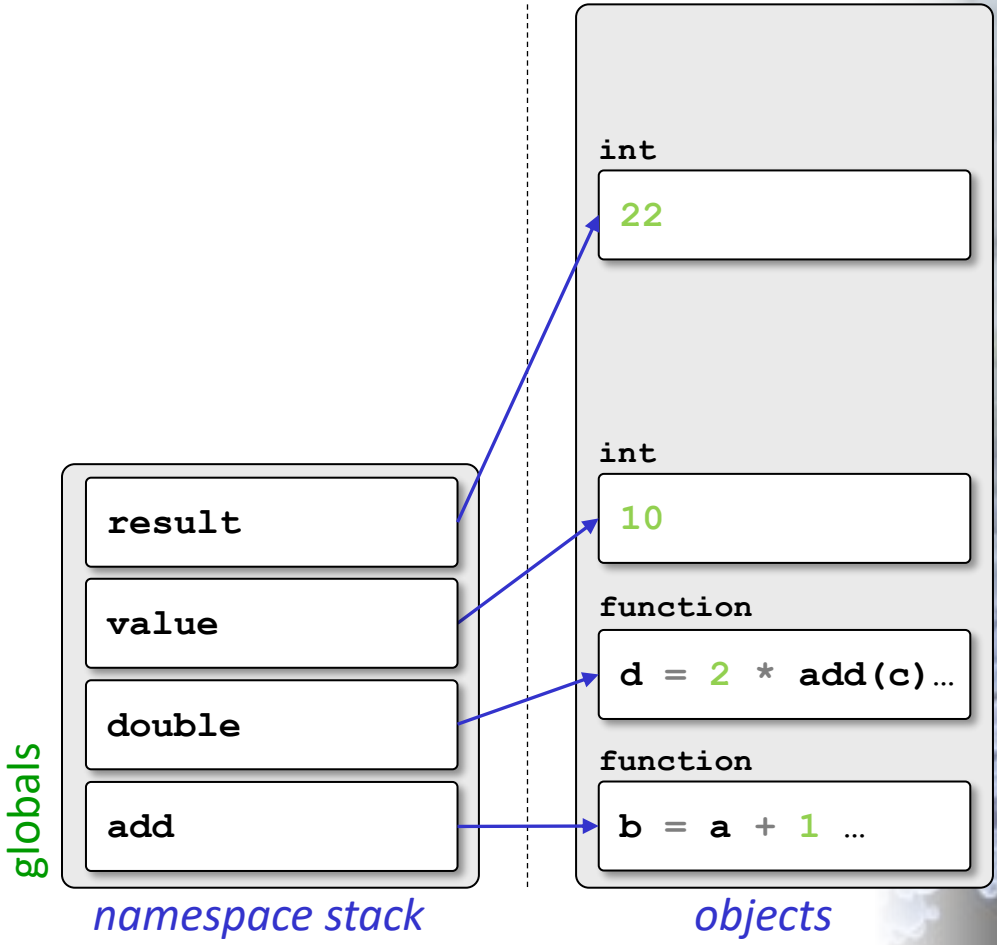
Calling functions



double.py

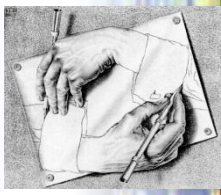
```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

each function call creates a new namespace for its local variables, and puts it on top of the namespace stack



UNIVERSITEIT GENT

Scope

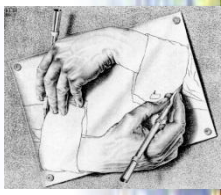


- **scope**: section of code in which a variable name is visible
  - depends on namespace stack
  - only names in top stack frame (*local variables*) and bottom stack frame (*global variables*) are visible

rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

type = 'sandstone'
result = classify(stone)
```

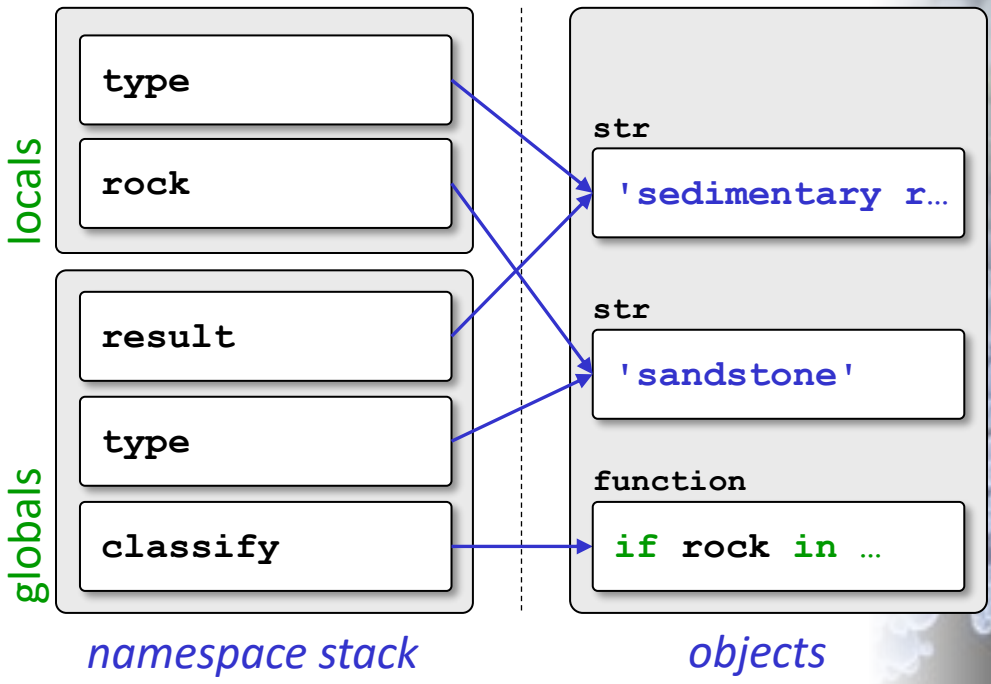
Scope



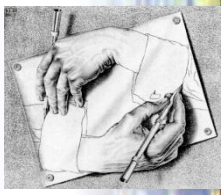
- **scope**: section of code in which a variable name is visible
  - depends on namespace stack
  - only names in top stack frame (*local variables*) and bottom stack frame (*global variables*) are visible
  - local scope takes precedence over global scope

rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

type = 'sandstone'
result = classify(stone)
```



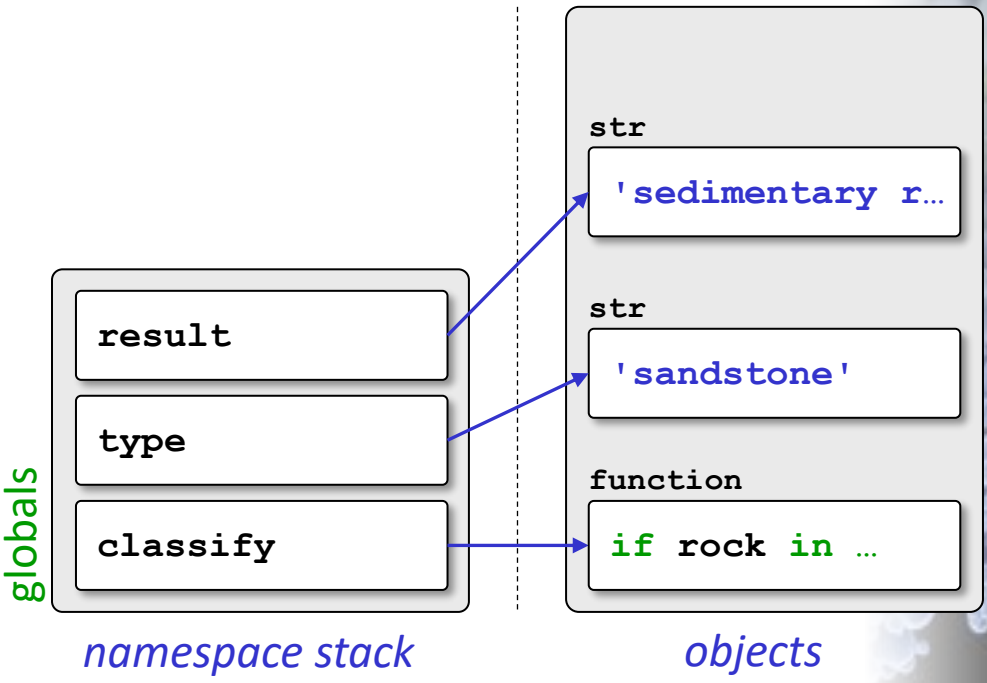
Scope



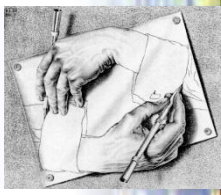
- **scope**: section of code in which a variable name is visible
  - depends on namespace stack
  - only names in top stack frame (*local variables*) and bottom stack frame (*global variables*) are visible
  - local scope takes precedence over global scope

rocks.py

```
def classify(rock):
    if rock in ['basalt', 'granite']:
        type = 'igneous rock'
    elif rock in ['sandstone', 'shale']:
        type = 'sedimentary rock'
    else:
        type = 'metamorphic rock'
    return type

type = 'sandstone'
result = classify(stone)
```



Pass and return expressions



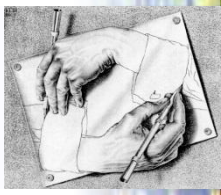
double.py

```
def add(a):
    b = a + 1
    return b

def double(c):
    d = 2 * add(c)
    return d

value = 10
result = double(value)
print(result)
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result

UNIVERSITEIT GENT

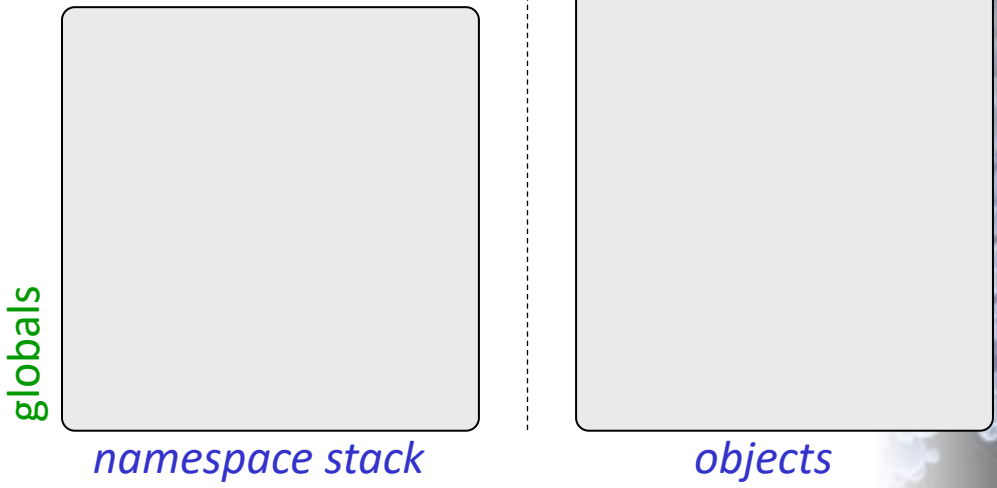
Pass and return expressions



double2.py

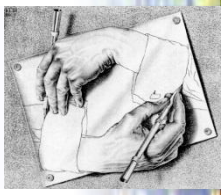
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

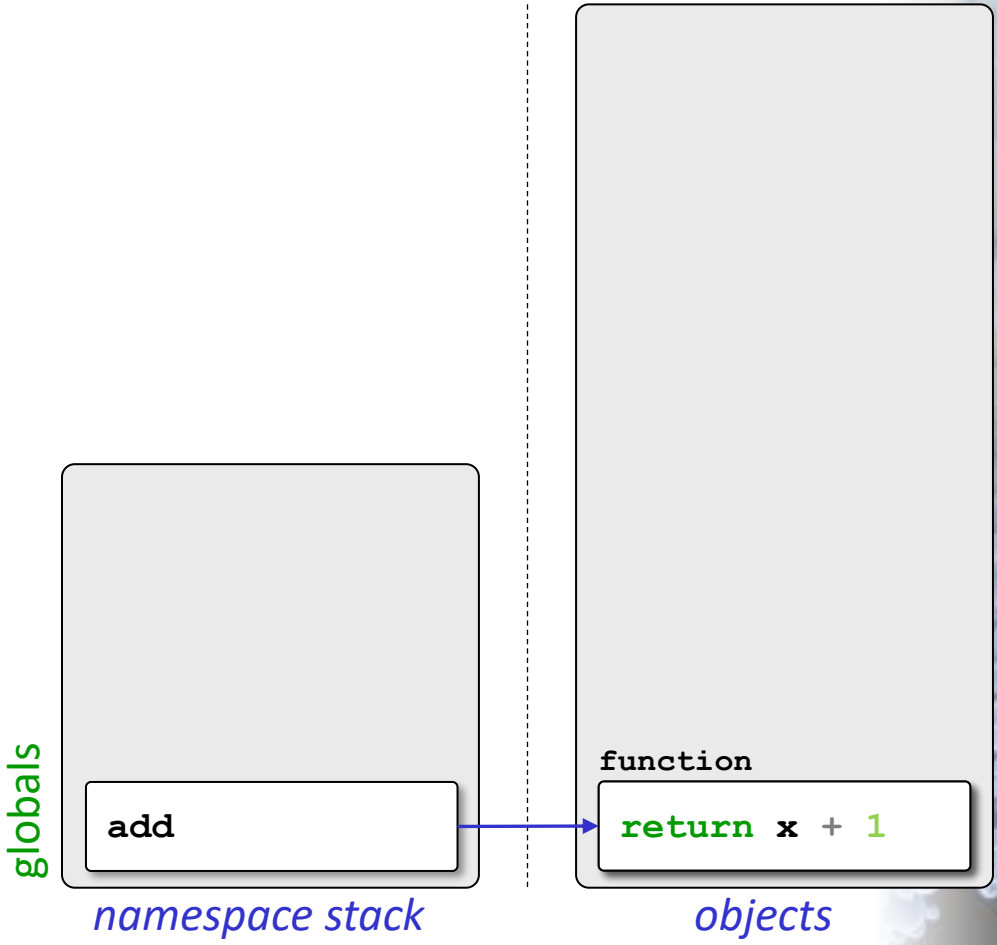
Pass and return expressions



double2.py

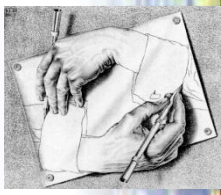
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

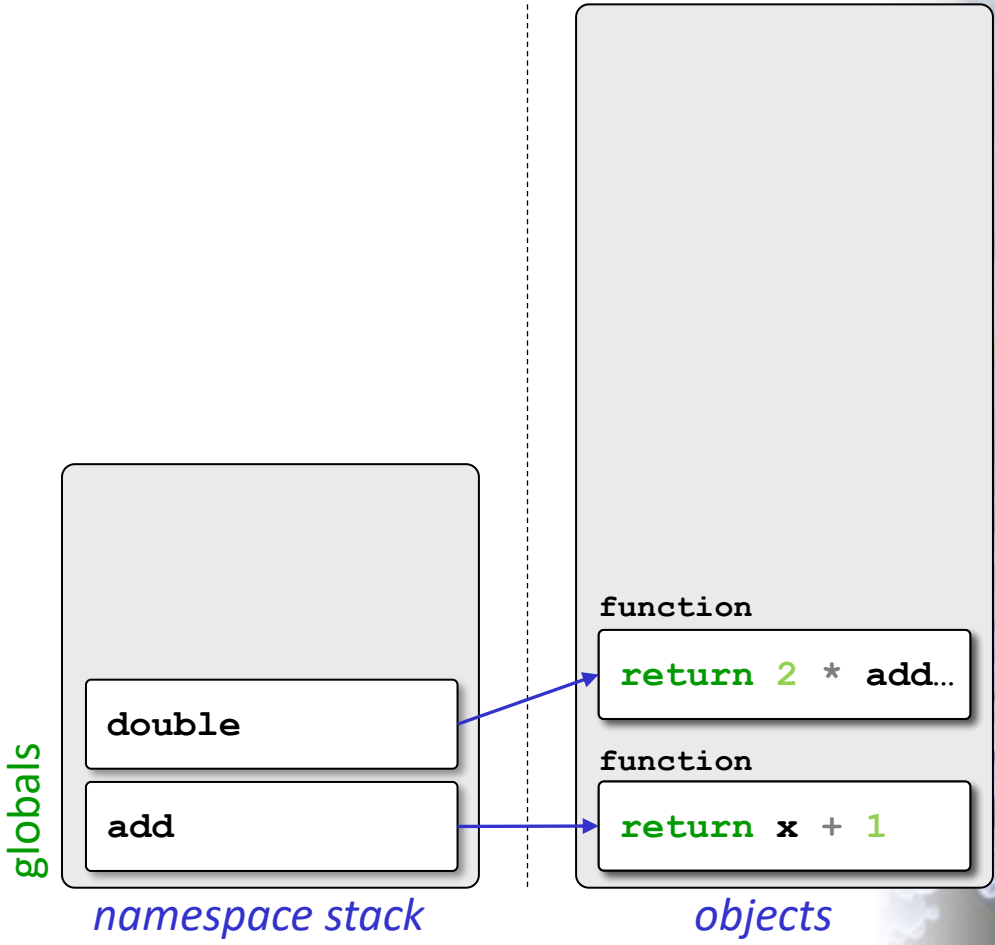
Pass and return expressions



double2.py

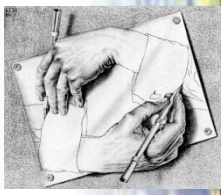
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

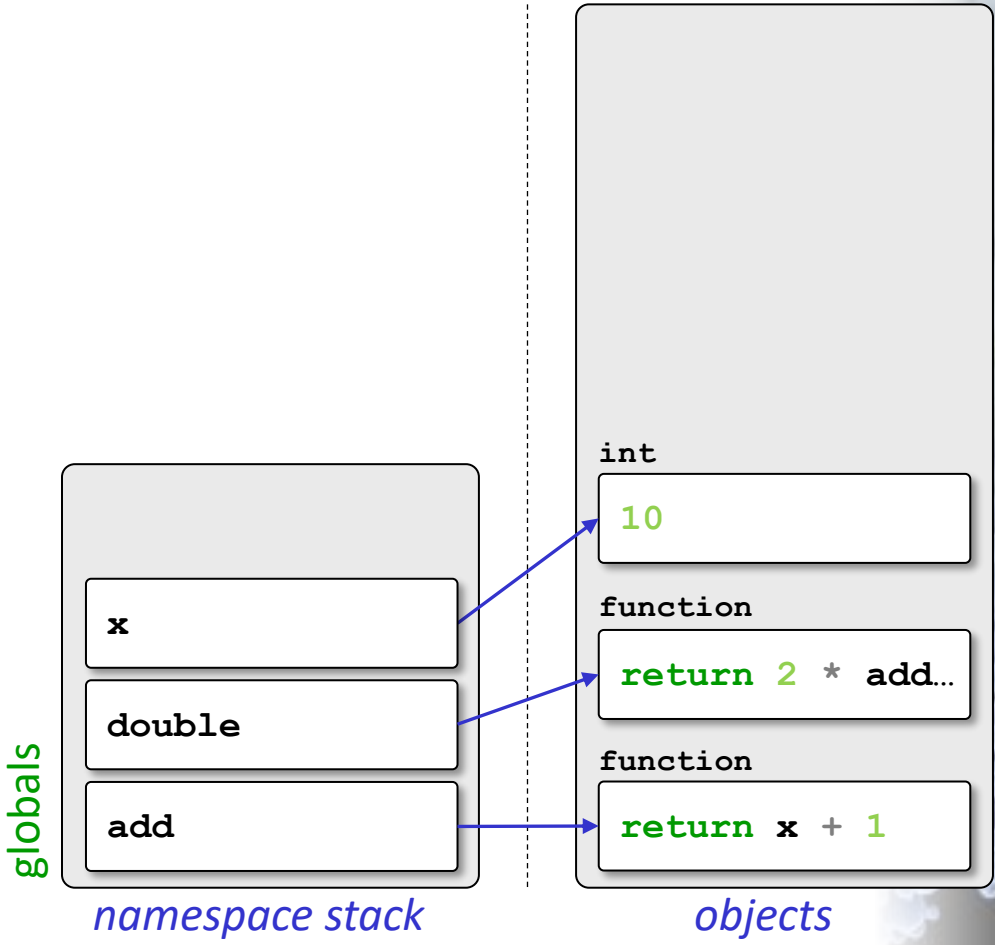
Pass and return expressions



double2.py

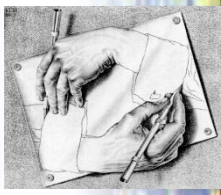
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

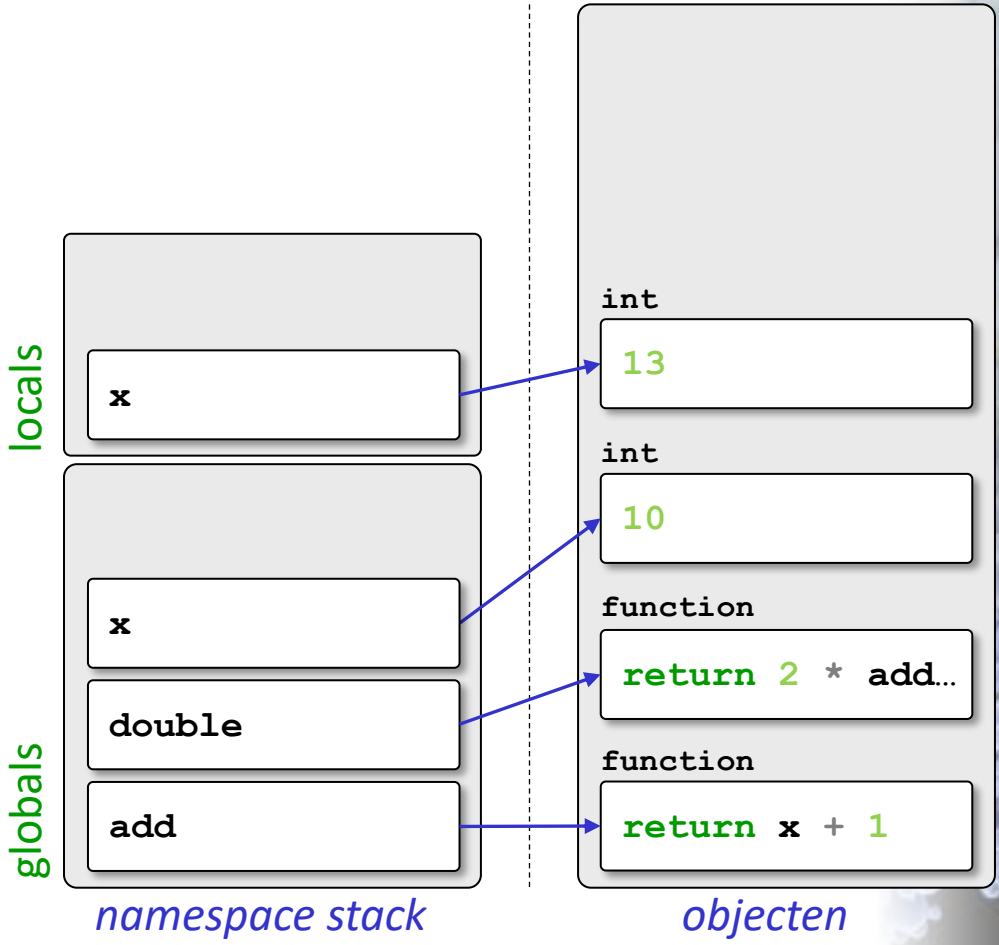
Pass and return expressions



double2.py

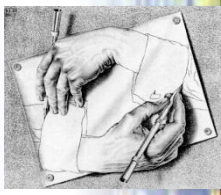
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

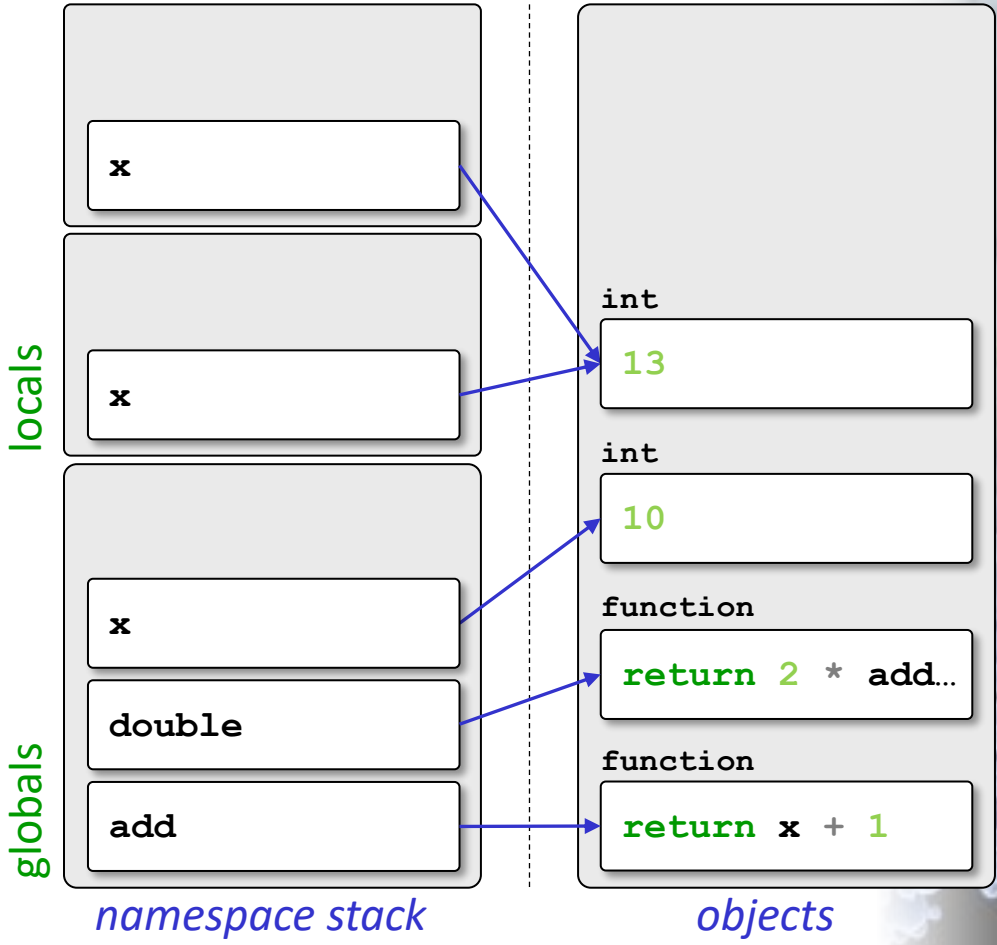
Pass and return expressions



double2.py

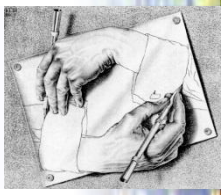
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

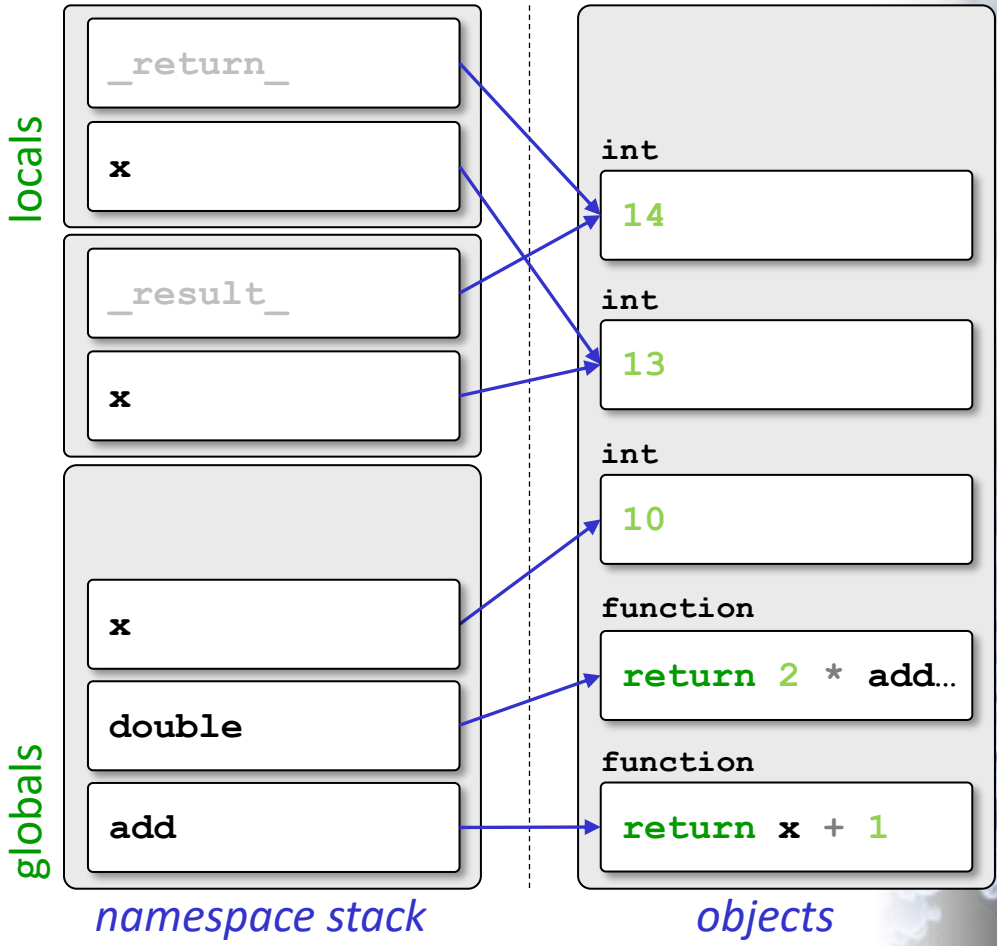
Pass and return expressions



double2.py

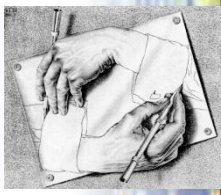
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

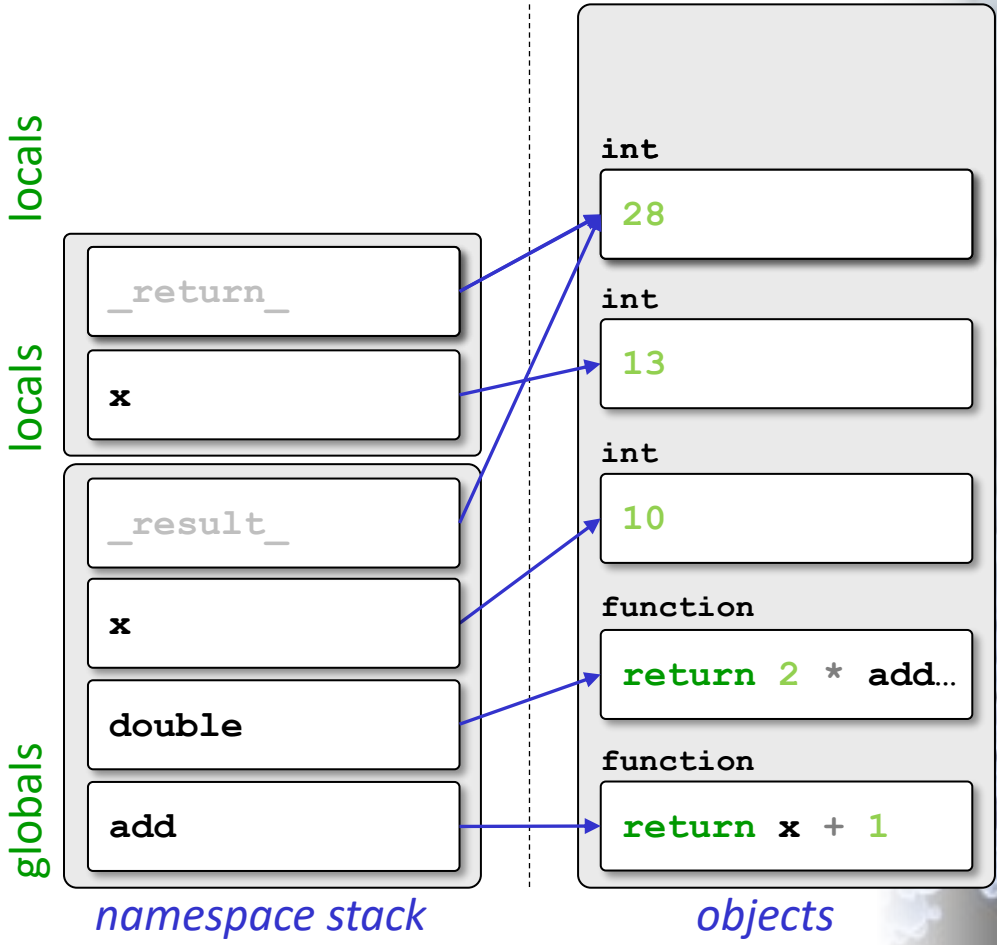
Pass and return expressions



double2.py

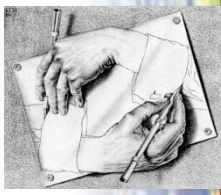
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

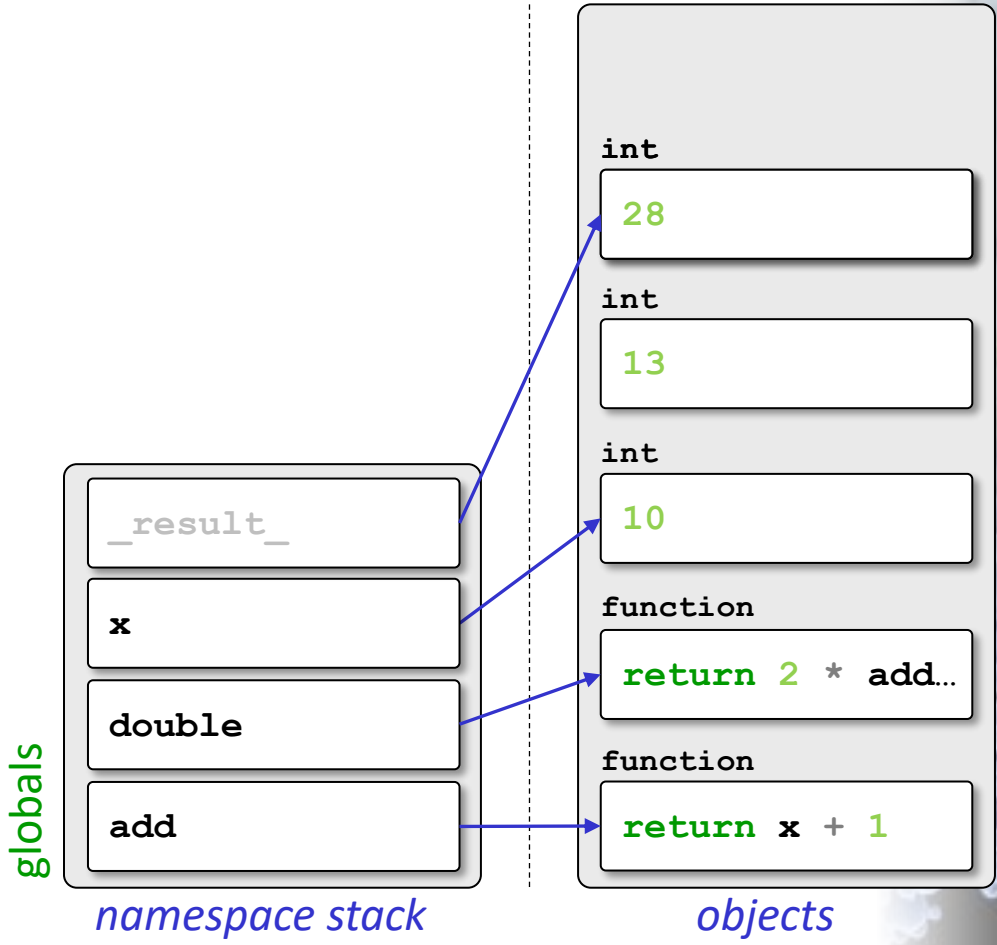
Pass and return expressions



double2.py

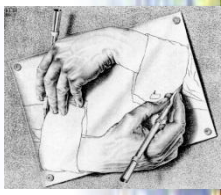
```
def add(x):
    return x + 1

def double(x):
    return 2 * add(x)

x = 10
print(double(x + 3))
```

an expression can be passed as an argument when calling a function, and a function can directly return an expression as its result



UNIVERSITEIT GENT

Returning objects



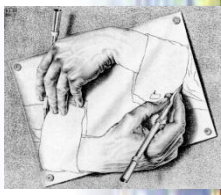
- return statement
  - evaluates expression and returns resulting object
  - afterwards the local scope of the function is removed from the namespace stack
- can occur everywhere in function body
- can occur multiple times in function body

sign.py

```
def sign(number):
    if number > 0:
        return 1
    elif number == 0:
        return 0
    else:
        return -1

print(sign(3))
```

UNIVERSITEIT GENT

Returning objects

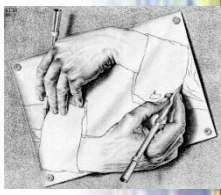


- return statement
  - evaluates expression and returns resulting object
  - afterwards the local scope of the function is removed from the namespace stack
- can occur everywhere in function body
- can occur multiple times in function body

sign.py

```
def sign(number):
    if number > 0:
        return 1
    elif number == 0:
        return 0
    else:
        return -1

print(sign(-9))
```

Returning objects



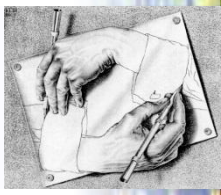
- over-use can make functions hard to understand
- no hard and fast rules for what's good practice, but …
  - small number of "early returns" at the very start of the function to handle special cases
  - one return at the end to return the "general" result

sign.py

```
def sign(number):
    if number > 0:
        return 1
    elif number == 0:
        return 0
    else:
        return -1

print(sign(-9))
```

UNIVERSITEIT GENT

Returning objects

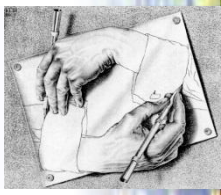


- *every* function returns something

sign.py

```
def sign(number):
    if number > 0:
        return 1
    elif number == 0:
        return 0
#    else:
#        return -1

print(sign(3))
```

Returning objects



- *every* function returns something
  - if a function does not explicitly return something else, it returns the value **`None`**

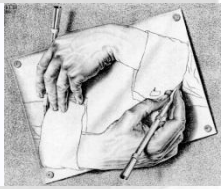
sign.py

```
def sign(number):
    if number > 0:
        return 1
    elif number == 0:
        return 0
#    else:
#        return -1

print(sign(-9))
```

`None`

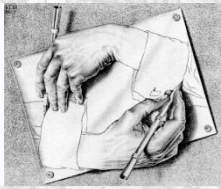
UNIVERSITEIT GENT

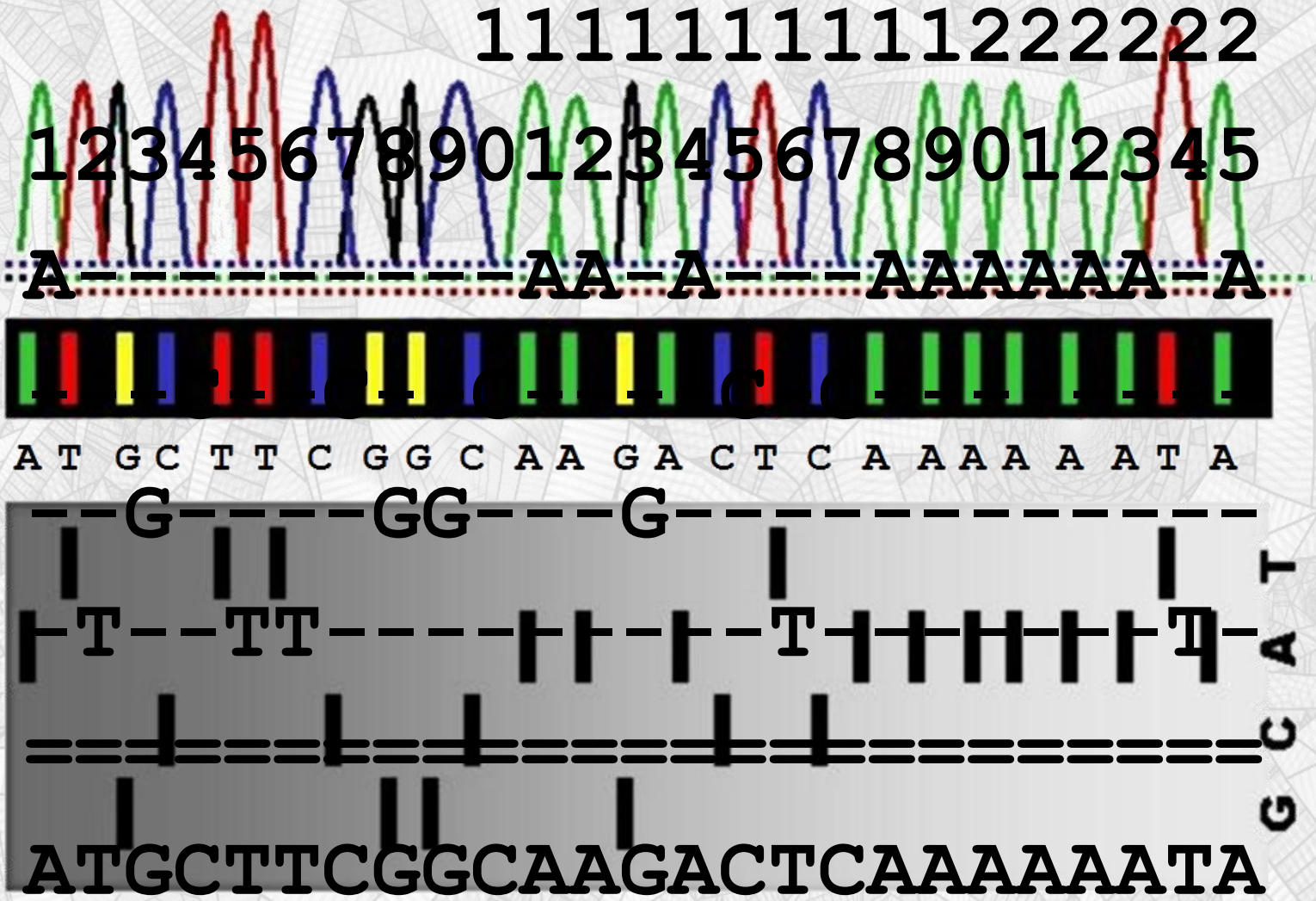
Emirp



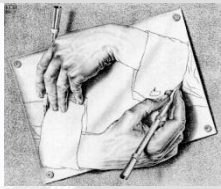
13, 17, 31, 37, 71, 73, 79, 97, 107, 113, 149, 157, 167, 179, 199, 311, 337, 347, 359, 389, 701, 709, 733, 739, 743, 751, 761, 769, 907, 937, 941, 953, 967, 971, 983, 991, 1009, 1021, 1031, 1033, 1061, 1069, 1091, 1097, 1103, 1109, 1151, 1153, 1181, 1193, …

Sanger sequencing

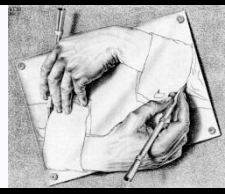




Mobile synonyms



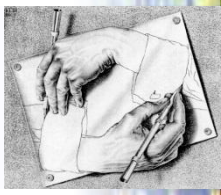
Sprigenroller



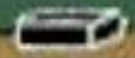
RDIAENG

Aoccdrnig to a rscheearch at an Elingsh uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are. The olny iprmoetnt tihng is taht the frist and lsat ltteer is at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae we do not raed ervey lteter by it slef but the wrod as a wlohe. ceehiro.

Homework (next lecture)

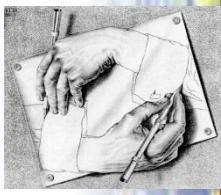


- course book
  - read chapter 7 (lists and tuples)
- read problem description of demo exercises
  - Chocolate game
  - Single-nucleotide polymorphism
  - Circadial permutator
- video tutorials
  - test-driven development
  - debugging source code



UNIVERSITEIT GENT

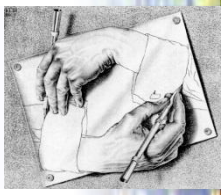
Homework (hands-on sessions)



- mandatory exercises of series 05 (functions)
  (deadline Tuesday, October 28, 2025, 22:00)
  - ISBN (demo)
  - Word sums
  - The last banana
  - Rear-view mirror
  - Inventory
  - Blinkenlights

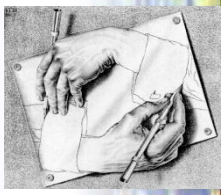
UNIVI
GENT

Questions or remarks



UNIVERSITEIT GENT

The sky is the limit …



"He who wonders, discovers
that this in itself is a wonder."

— MC Escher

UNIVERSITEIT
GENT