Ghent University
Faculty of Sciences

Python Programming

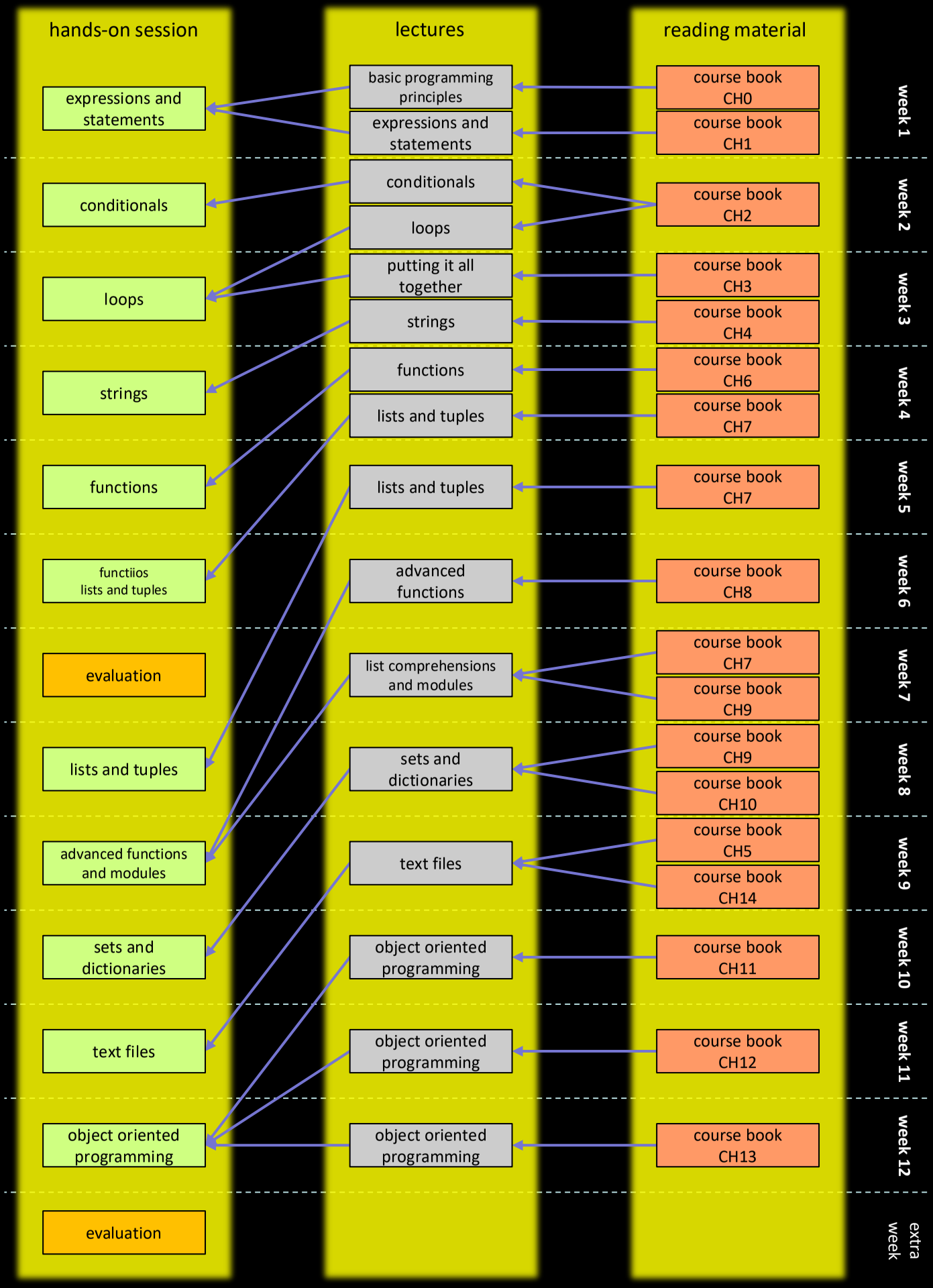
list comprehensions and modules

Prof. Dr. Peter Dawyndt

peter.dawyndt@ugent.be

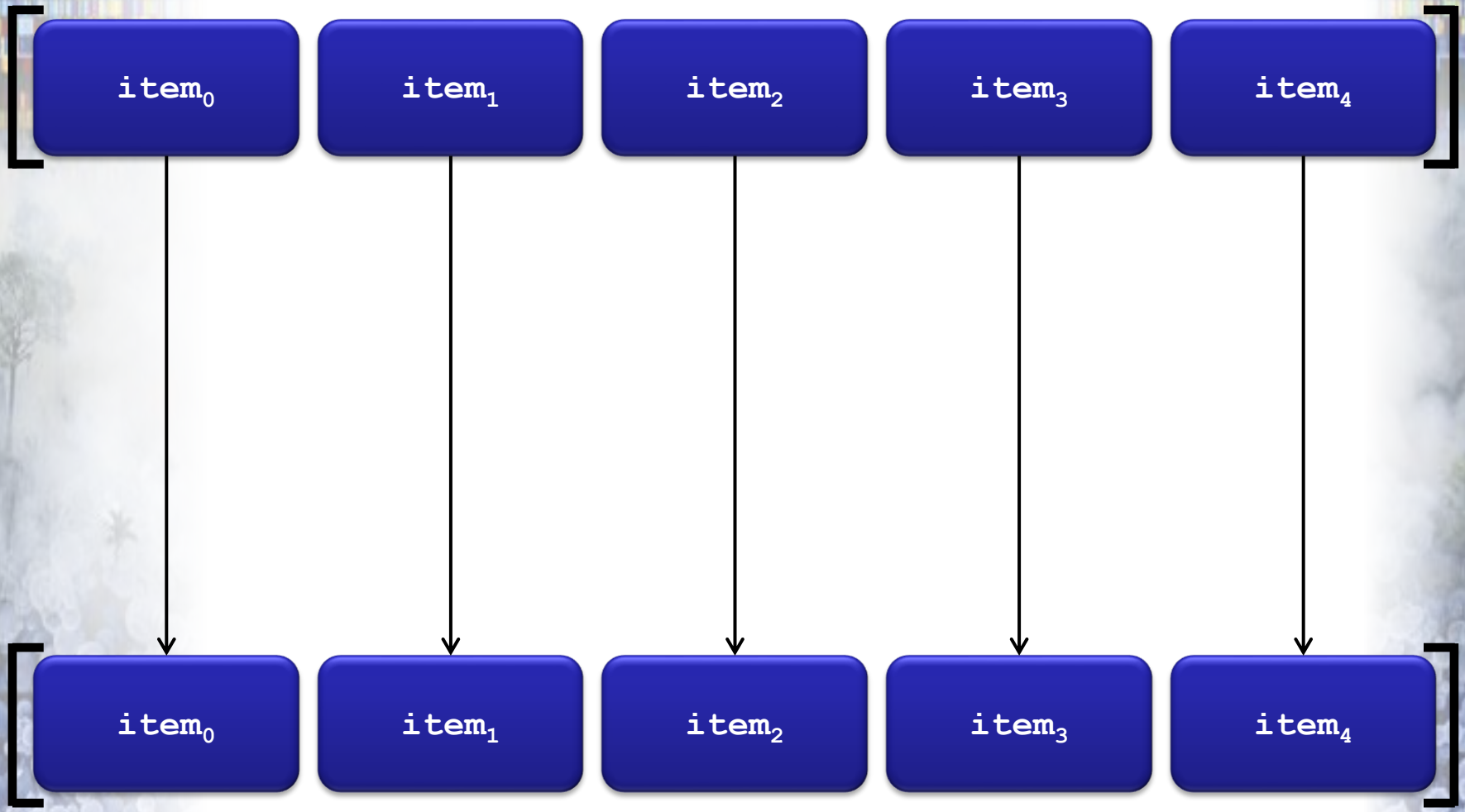
@dawyndt

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

List comprehensions

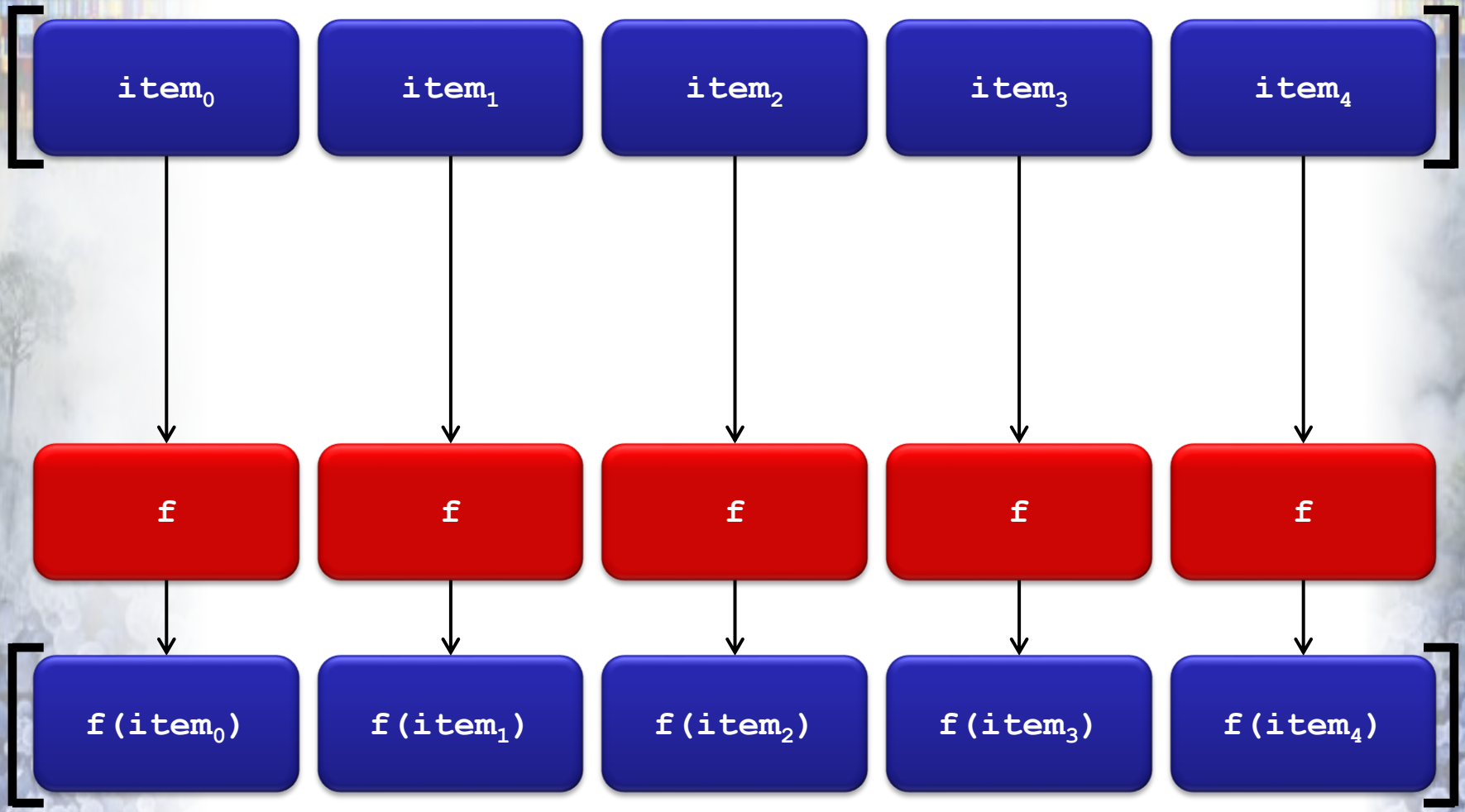
List comprehensions



UNIVERSITEIT GENT

```
newlist = [item for item in oldlist]
```

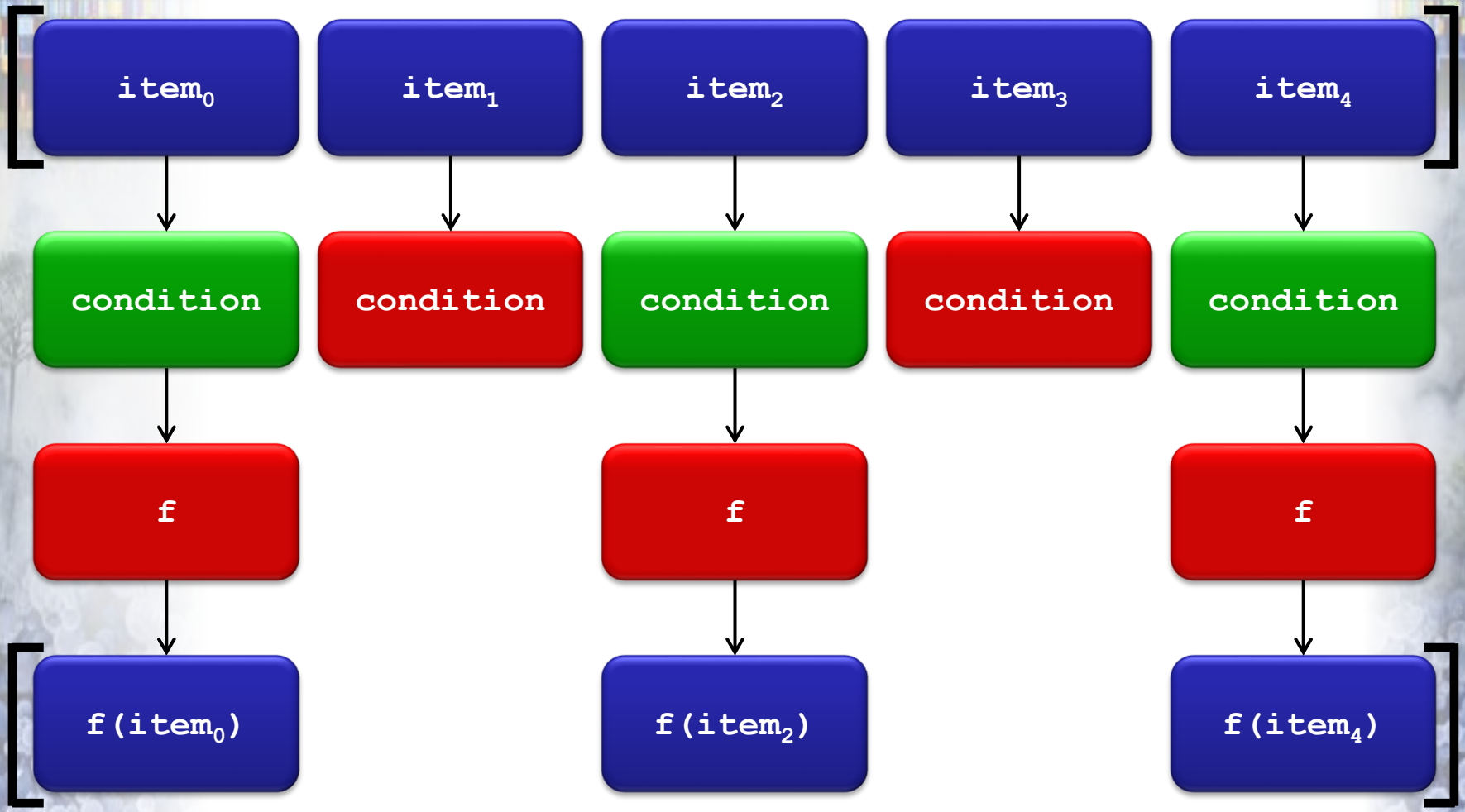
List comprehensions



UNIVERSITEIT GENT

```
newlist = [f(item) for item in oldlist]
```

List comprehensions



UNIVERSITEIT GENT

```
newlist = [f(item) for item in oldlist if condition]
```

List comprehensions

- workhorse data type in Python
  - creates new list by applying expression to elements of an initial list
  - heavily used by Python programmers (*pythonic*)
    - lots of examples in code fragments found online
- unusual syntax
  - similar to syntax of **`for`** loop, **`in`** operator and **`if`** statement
  - same three keywords (**`for`**, **`in`** and **`if`**) are also used (sometimes even repeatedly) in list comprehensions

UNIVERSITEIT GENT

List comprehensions

- basic format

```
[ expression for variable in iterable ]
```

  - ***expression*** is expression applied to ***variable***
  - for each element of ***iterable*** the list comprehension will
    1. reference element with ***variable***
    2. compute new value as evaluation of ***expression***
       - tuple expressions must be enclosed with round brackets
  - computed values are bundled into a new list that is returned as the result of the list comprehension

```
>>> li = [3, 6, 2, 7]
>>> [2 * elem for elem in li]
[6, 12, 4, 14]
```

UNIVERSITEIT GENT

List comprehensions

- basic format

```
[ expression for variable in iterable ]
```

  - ***iterable*** may contain elements of heterogeneous types
    - ***expression*** must be valid for each element of ***iterable***
  - elements of ***iterable*** can also be containers themselves
    - ***variable*** is container of variables with data type and "form" that corresponds to elements of ***iterable***

```
>>> li = [3, 'spam', [4, 5]]
>>> [3 * elem for elem in li]
[9, 'spamspamspam', [4, 5, 4, 5, 4, 5]]
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [3 * n for (x, n) in li]
[3, 6, 21]
```

UNIVERSITEIT GENT

List comprehensions

- basic format

```
[ expression for variable in iterable ]
```

  - ***expression*** may contain user-defined function
    - ***expression*** must be valid for each element of ***iterable***

```
>>> def subtract(a, b):
...     return a - b
...
>>> li = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in li]
[-3, 6, 0]
```

UNIVERSITEIT GENT

Additional filtering

- extended format

```
[ expression for variable1 in iterable1
  [ if condition | for variable2 in iterable2 ] … ]
```

  - ***condition*** determines whether or not ***expression*** is applied to individual elements of ***iterable***
    - ***condition*** is tested for each element of ***iterable*** in the preceding ***for*** clause of the list comprehension
    - elements for which ***condition*** is **False** are left out of resulting list before evaluation of **expressie** from the list comprehension

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [2 * elem for elem in li if elem > 4]
[12, 14, 18]
```

UNIVERSITEIT GENT

Additional filtering

```
>>> '1 2 3 4'.split()
['1', '2', '3', '4']
>>> [int(x) for x in '1 2 3 4'.split()]   # conversion
[1, 2, 3, 4]
>>> [x * x for x in range(10)]            # power
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x for x in range(10) if not x % 2]   # odd numbers
[0, 2, 4, 6, 8]
>>> [(b, a) for a, b in [(1, 2), (3, 4)]] # exchange
[(2, 1), (4, 3)]
>>> s = "monty python's flying circus"
>>> [ord(c) for c in s]
[109, 111, 110, 116, 121, 32, 112, 121, 116, 104, … ]
>>> [i for i, c in enumerate(s) if c == ' ']
[5, 14, 21]
>>> [p for p in range(2, 100)
...      if not [x for x in range(2, p) if not p % x]]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, …, 97]
```

UNIVERSITEIT GENT

Nesting list comprehensions

- list comprehension takes an iterable as input and returns a list as output
  - list is itself an iterable
  - enables possibility to nest list comprehensions

- example
  - inner comprehension results in list `[4, 3, 5, 2]`
  - outer comprehension results in list `[8, 6, 10, 4]`

```
>>> li = [3, 2, 4, 1]
>>> [2 * elem for elem in [item + 1 for item in li]]
[8, 6, 10, 4]
```

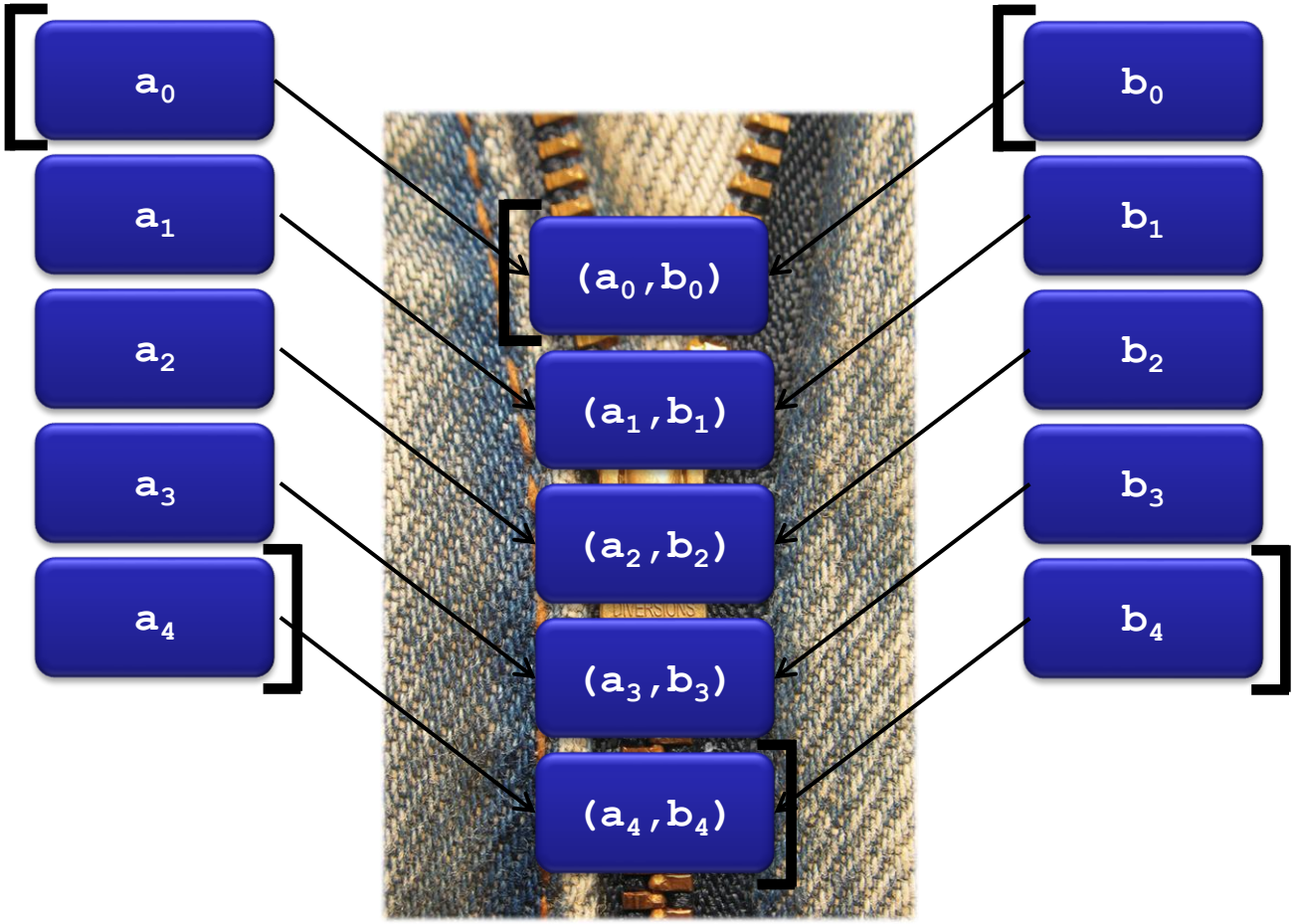
UNIVERSITEIT GENT

Nesting list comprehensions

```
>>> vec = [2, 4, 6]
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]    # tuple requires round brackets
  File "<stdin>", line 1
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x * y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x + y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i] * vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

UNIVERSITEIT GENT

Zip



UNIVERSITEIT GENT

Zip

- combines two or more iterable objects by aggregating elements at corresponding positions into tuples that are the elements of a new iterable

```
>>> firstnames = ['John', 'Terry', 'Eric']
>>> lastnames = ['Cleese', 'Gilliam', 'Idle']
>>> names = list(zip(firstnames, lastnames))
>>> names
[('John', 'Cleese'), ('Terry', 'Gilliam'), ('Eric', 'Idle')]
>>> list(zip(*names))      # also works for multiple lists
[('John', 'Terry', 'Eric'), ('Cleese', 'Gilliam', 'Idle')]
                           # * before argument results in
                           # zip(names[0],names[1],names[2])
```

UNIVERSITEIT GENT

Modules

Modules

- **module**: file containing Python definitions and statements
  - goal: reuse definitions in other Python code
  - **The Python Standard Library**
    - library of modules that is shipped together with Python
    - "batteries included"
  - Python + standard library + scientific modules
    - Enthought Python Distribution (EDP)
    - Anaconda
  - and many more modules…
    - Python Package Index (aka "The Cheese Shop")

"Don't reinvent the wheel."

UNIVERSITEIT GENT

python™

ANACONDA

ENTHOUGHT SCIENTIFIC COMPUTING SOLUTIONS

BATTERIES INCLUDED

Creating modules

- every Python file is automatically also a module
  - brought into scope using the **`import`** statement
  - module are also objects having properties and methods
  - access properties and methods with a dot (*dot operator*)

circle.py

```
"""Working with circles."""

from math import pi

def perimeter(radius):
    """Perimeter of a circle."""
    return 2 * pi * radius

def area(radius):
    """Area of a circle."""
    return pi * radius ** 2
```

```
>>> radius = 2.87
>>> import circle
>>> type(circle)
<class 'module'>
>>> dir(circle)
['__builtins__', '__cached__', '__doc__',
 '__file__', '__name__', '__package__', 'pi',
 'perimeter', 'area']
>>> circle.perimeter(radius)
18.0327418316
>>> circle.area(radius)
25.8769845284
```

Creating modules

- every Python file is automatically also a module
  - brought into scope using the **`import`** statement
  - module are also objects having properties and methods
  - access properties and methods with a dot (*dot operator*)

circle.py

```
"""Working with circles."""

from math import pi

def perimeter(radius):
    """Perimeter of a circle."""
    return 2 * pi * radius

def area(radius):
    """Area of a circle."""
    return pi * radius ** 2
```

```
>>> import circle
>>> circle.__name__
'circle'
>>> circle.__doc__
'Working with circles.'
>>> help(circle)
Help on module circle:

NAME
    circle - Working with circles.

FUNCTIONS
    perimeter(radius)
        Perimeter of a circle.
    area(radius)
        Area of a circle.
```

docstring

Creating modules

- every Python file is automatically also a module
  - brought into scope using the **`import`** statement
  - module are also objects having properties and methods
  - access properties and methods with a dot (*dot operator*)

circle.py

```
"""Working with circles."""

from math import pi

def perimeter(radius):
  """Perimeter of a circle."""
  return 2 * pi * radius

def area(radius):
  """Area of a circle."""
  return pi * radius ** 2
```

analysis.py

```
import circle

radius = input('radius: ')

ftest = radius
if '.' in radius:  # float given ?
  ftest = radius.replace('.', '', 1)

if ftest.isdigit():
  radius = float(radius)
  print(circle.perimeter(radius))
  print(circle.area(radius))
```

Module scope

- **namespace**: syntactic container which permits same name to be used in different modules and functions (and classes)
- each module determines its own namespace
  - same name in multiple modules causes no identification problem

module1.py

```
question = 'What is the meaning of life, the Universe, and everything?'
answer = 42
```

module2.py

```
question = 'What is your quest?'
answer = 'To seek the holy grail.'
```

```
>>> import module1
>>> import module2
>>> module1.question
'What is the meaning of life, the Universe, and everything?'
>>> module2.question
'What is your quest?'
```

UNIVERSITEIT GENT

Module scope

- **namespace**: syntactic container which permits same name to be used in different modules and functions (and classes)
- each module determines its own namespace
  - same name in multiple modules causes no identification problem

module1.py

```
question = 'What is the meaning of life, the Universe, and everything?'
answer = 42
```

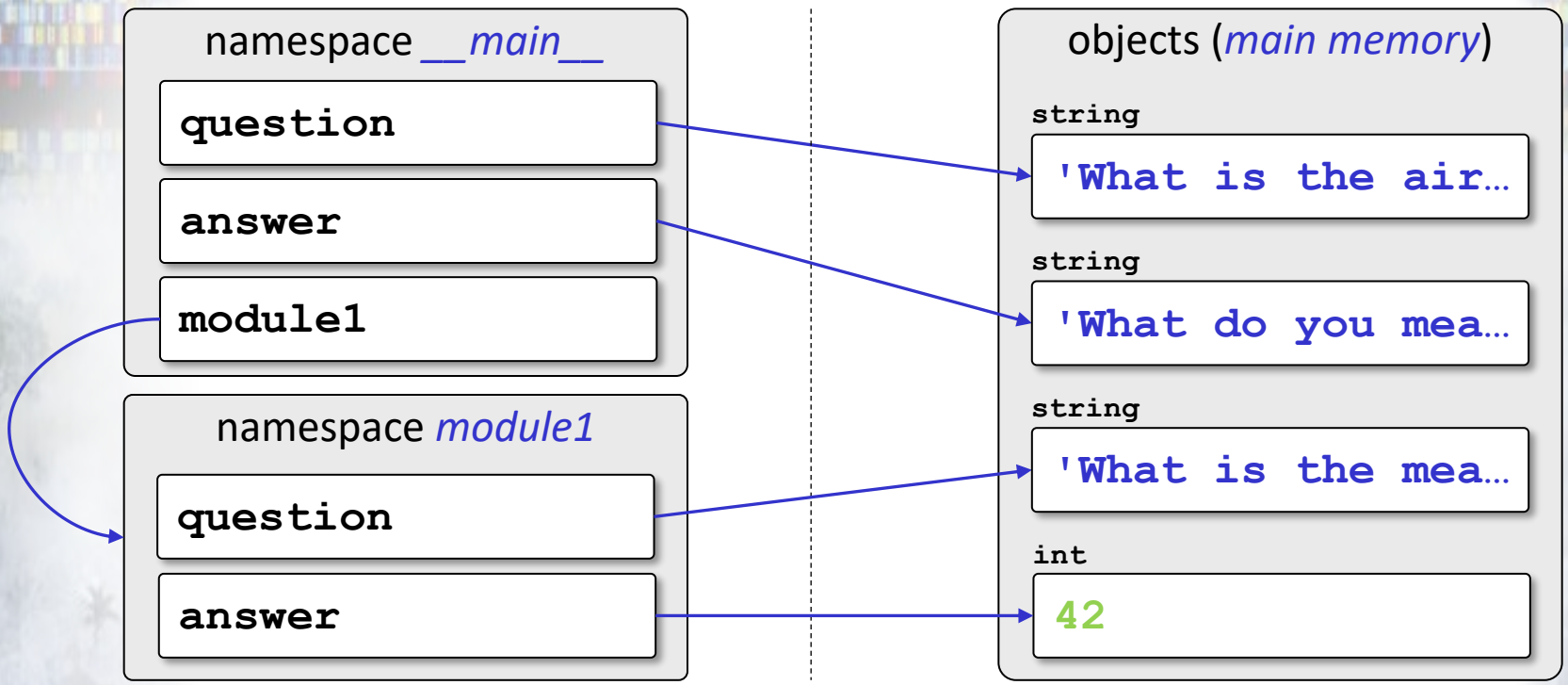
module2.py

```
question = 'What is your quest?'
answer = 'To seek the holy grail.'
```

```
>>> module1.answer
42
>>> module2.answer
'To seek the holy grail.'
```

UNIVERSITEIT GENT

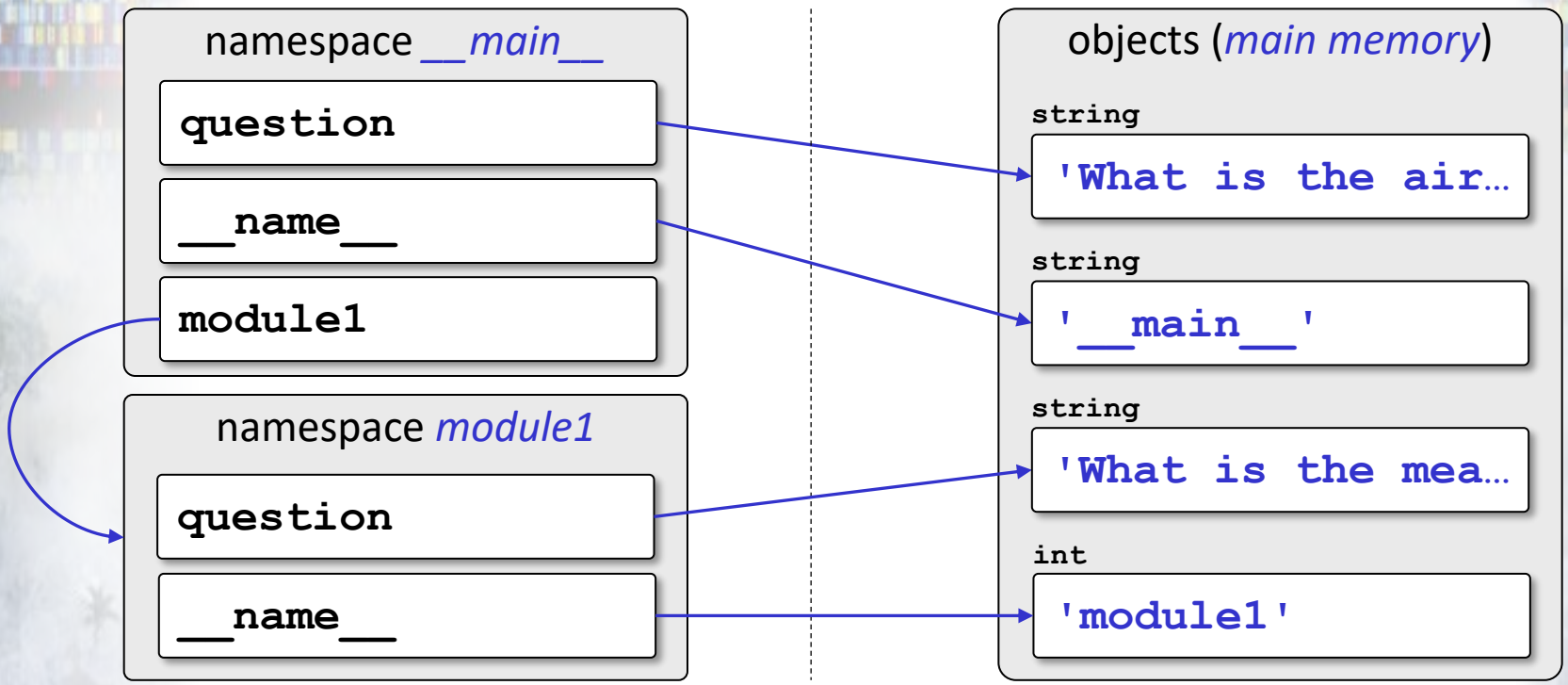
Module scope




```
>>> question = 'What is the air-speed velocity of an unladen swallow?'
>>> answer = 'What do you mean? An African or European swallow?'
>>> import module1
>>> question
'What is the air-speed velocity of an unladen swallow?'
>>> module1.question
'What is the meaning of life, the Universe, and everything?'
```

UNIVERSITEIT GENT

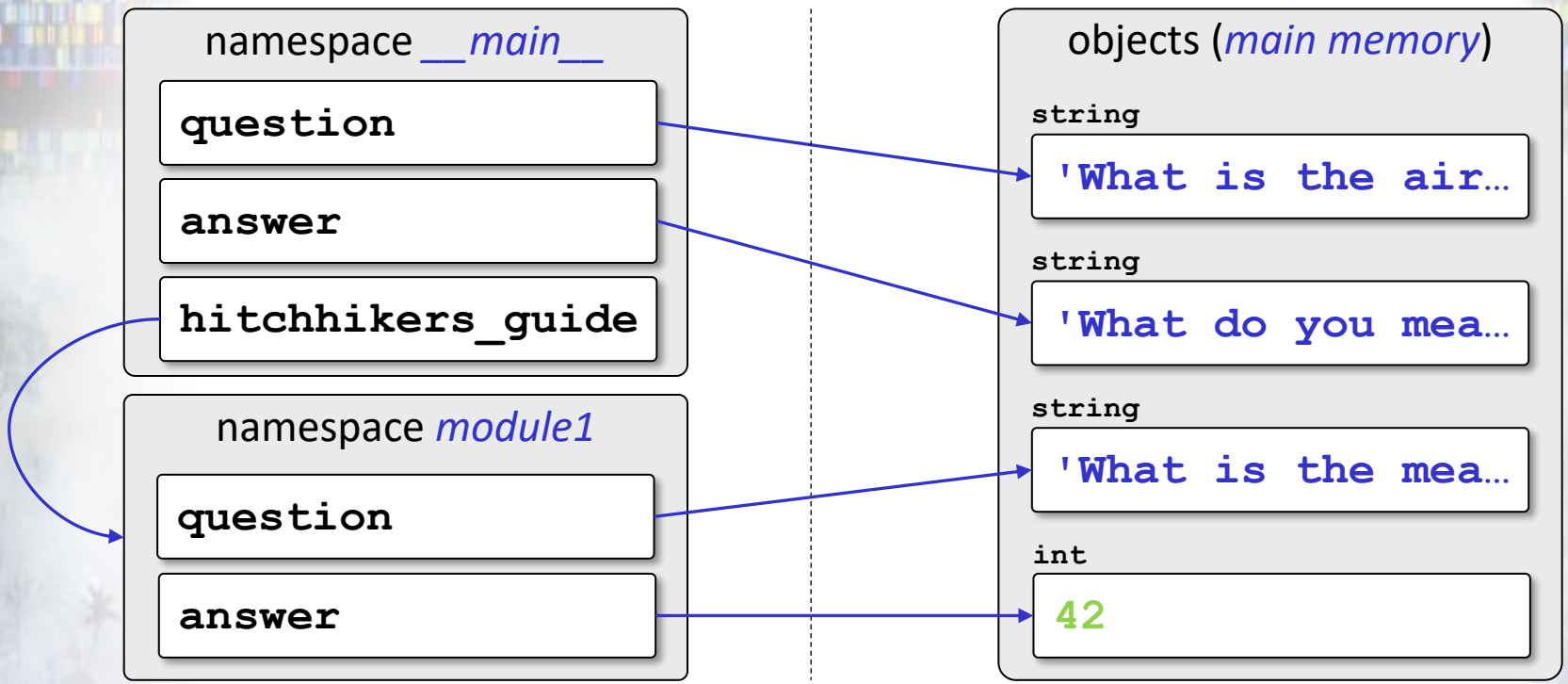
Module scope




```
>>> import module1
>>> __name__
'__main__'
>>> module1.__name__
'module1'
```

UNIVERSITEIT GENT

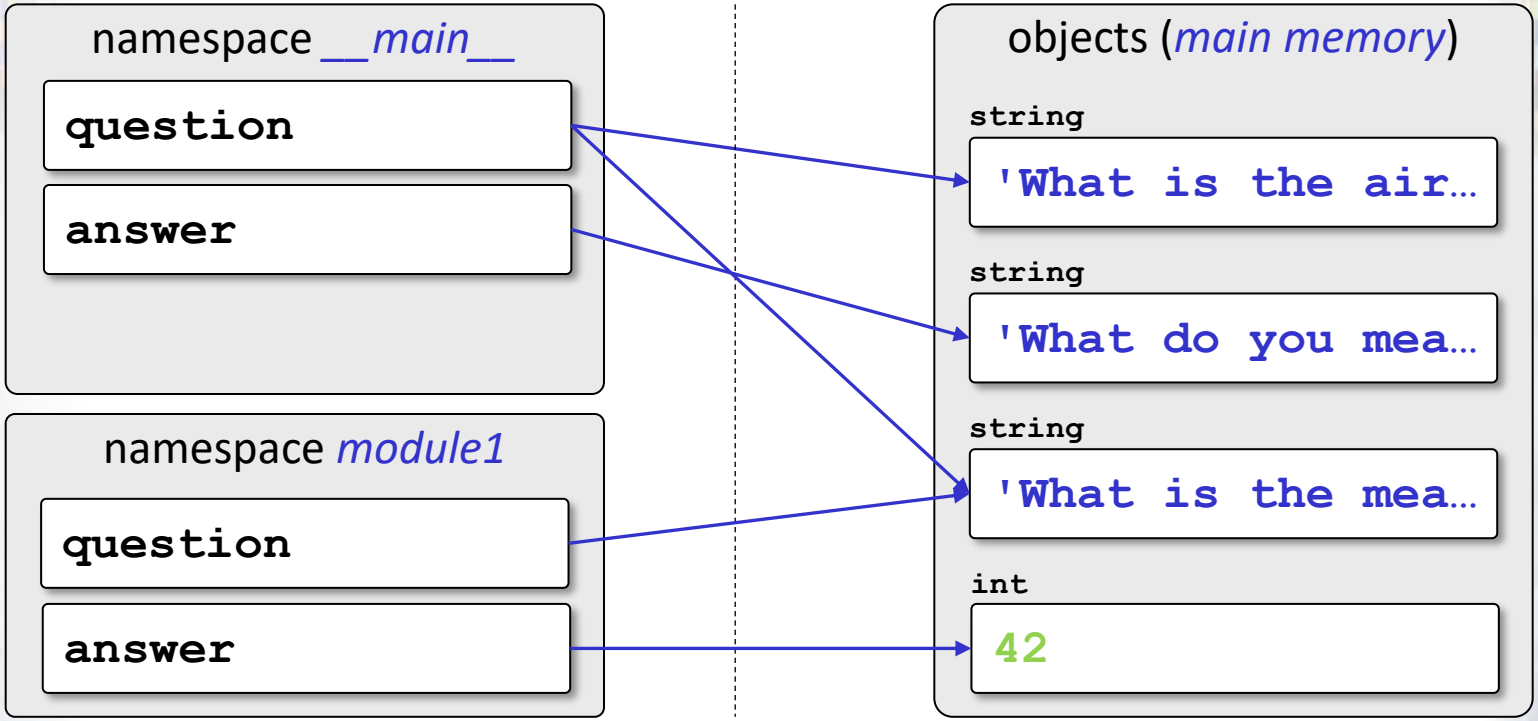
Alternative imports




```
>>> question = 'What is the air-speed velocity of an unladen swallow?'
>>> answer = 'What do you mean? An African or European swallow?'
>>> import module1 as hitchhikers_guide
>>> question
'What is the air-speed velocity of an unladen swallow?'
>>> hitchhikers_guide.question
'What is the meaning of life, the Universe, and everything?'
```

UNIVERSITEIT GENT

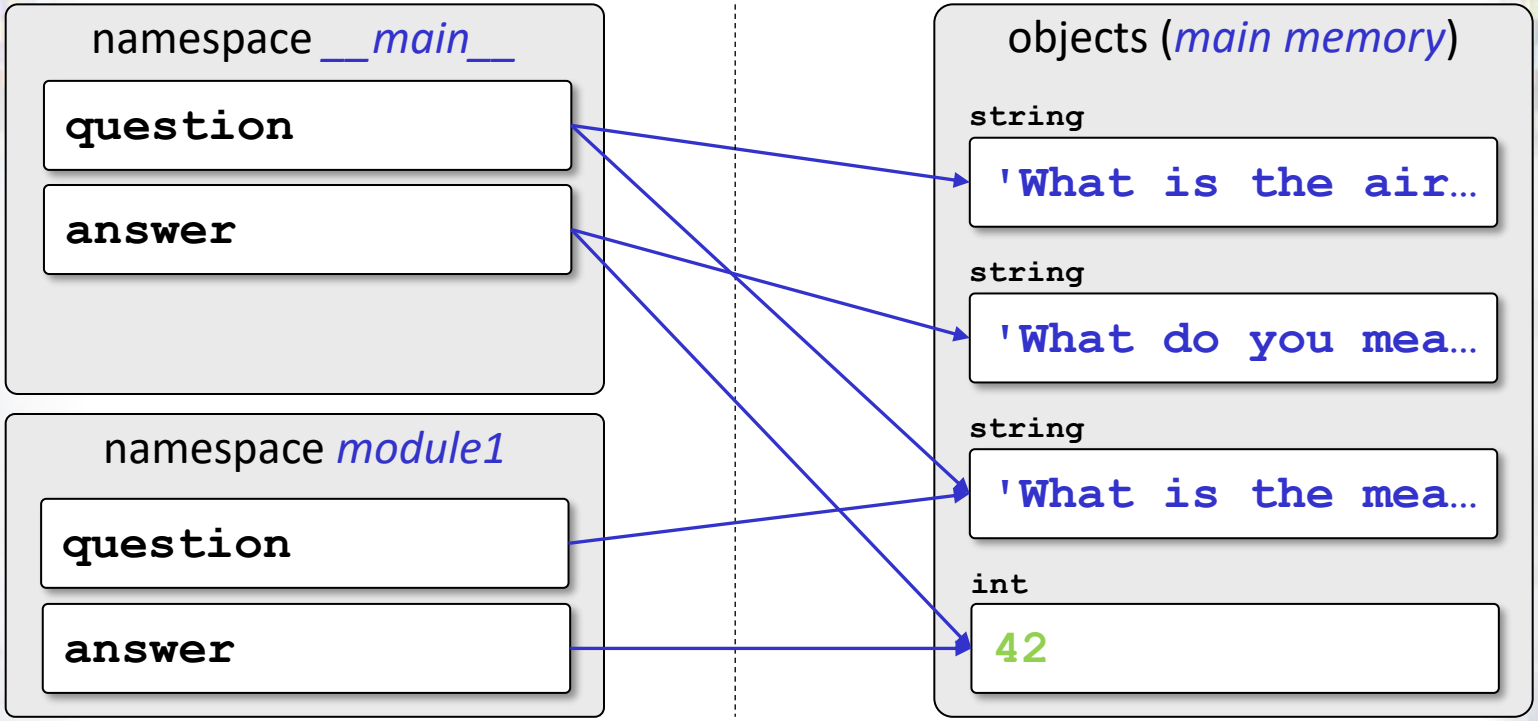
Alternative imports




```
>>> question = 'What is the air-speed velocity of an unladen swallow?'
>>> answer = 'What do you mean? An African or European swallow?'
>>> from module1 import question
>>> question
'What is the meaning of life, the Universe, and everything?'
>>> answer
'What do you mean? An African or European swallow?'
```

UNIVERSITEIT GENT

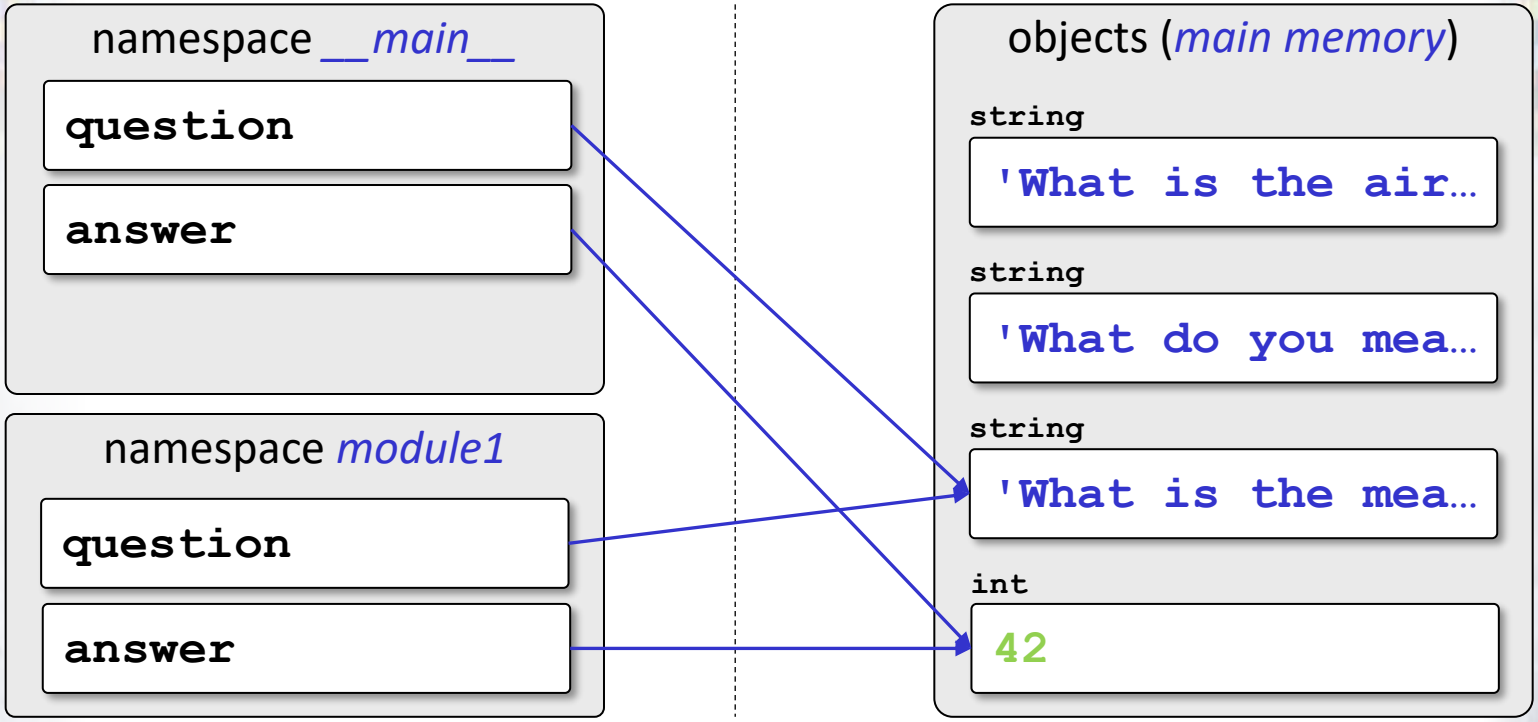
Alternative imports




```
>>> question = 'What is the air-speed velocity of an unladen swallow?'
>>> answer = 'What do you mean? An African or European swallow?'
>>> from module1 import question, answer
>>> question
'What is the meaning of life, the Universe, and everything?'
>>> answer
42
```

UNIVERSITEIT GENT

Alternative imports




```
>>> question = 'What is the air-speed velocity of an unladen swallow?'
>>> answer = 'What do you mean? An African or European swallow?'
>>> from module1 import *
>>> question
'What is the meaning of life, the Universe, and everything?'
>>> answer
42
```

"Almost always a bad idea."

UNIVERSITEIT GENT

Module scope

- **namespace**: syntactic container which permits same name to be used in different modules and functions (and classes)
- functions also have their own namespace

namespace.py

```
def f():
    n = 7
    print(f'n inside f: {n}')

def g():
    n = 42
    print(f'n inside g: {n}')

n = 11
print(f'n before calling f: {n}')
f()
print(f'n after calling f: {n}')
g()
print(f'n after calling g: {n}')
```

```
$ python namespace.py
n before calling f: 11
n inside f: 7
n after calling f: 11
n inside g: 42
n after calling g: 11
$
```

UNIVERSITEIT GENT

Import executes statements

- **`import`** is a statement
  - executed when Python encounters it
  - just like any other statement
- statements in module are executed as it is loaded
  - assignment and **`def`** are statements
  - conditionals, loops and anything else can also be used
- names are imported into current namespace

UNIVERSITEIT GENT

Import executes statements

```
>>> import three_questions
What is your name?
Sir Galahad of Camelot.
What is your quest?
I seek the Holy Grail.
What is your favorite color?
Blue.  No yell--
Auuuuuuuugh!
>>>
```

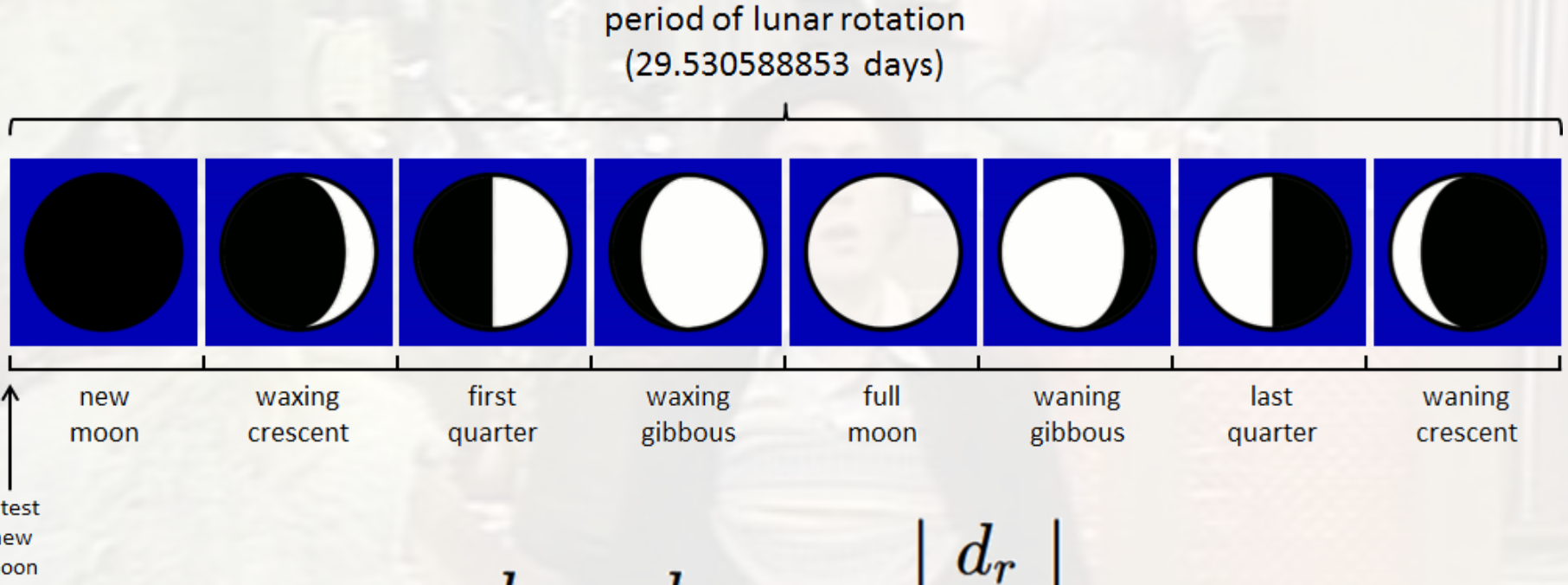
three_questions.py

```
answer1 = input('What is your name?\n')
answer2 = input('What is your quest?\n')
answer3 = input('What is your favorite color?\n')
if answer3.lower()[:6] == 'yellow':
    print('Right. Off you go.')
else:
    print('Auuuuuuuugh!')
```

UNIVERSITEIT GENT

Lunar phases

Lunar phases



$$d_n = d_r - p * \left\lfloor \frac{d_r}{p} \right\rfloor$$

number of days since latest new moon

number of days since reference date (January 6, 2000)

Test-driven development

Test-driven development (TDD) is a software development practice which arrives at a desired feature through a series of small, iterative steps motivated by automated tests (which are *written first*) that express increasing refinements of the desired feature.

Doctests

matrix1.py

```
def makeMatrix(rows, columns):
    """
    Returns a matrix with a given number of rows and
    columns. All matrix elements are initialized to 0.

    >>> makeMatrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

```
>>> from matrix1 import makeMatrix
>>> makeMatrix(3, 5)
>>> print(makeMatrix(3, 5))
None
>>>
```

UNIVERSITEIT GENT

Doctests

```
$ python matrix1.py
**********************************************************************
File "matrix1.py", line 6, in __main__.makeMatrix
Failed example:
    makeMatrix(3, 5)
Expected:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
Got nothing
**********************************************************************
1 items had failures:
   1 of   1 in __main__.makeMatrix
***Test Failed*** 1 failures.
$
```

UNIVERSITEIT GENT

Doctests

matrix2.py

```
def makeMatrix(rows, columns):
    """
    Returns a matrix with a given number of rows and
    columns. All matrix elements are initialized to 0.

    >>> makeMatrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """
    return [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

```
$ python matrix2.py
$
```

UNIVERSITEIT GENT

Doctests

matrix3.py

```
def makeMatrix(rows, columns):
    """
    Returns a matrix with a given number of rows and
    columns. All matrix elements are initialized to 0.

    >>> makeMatrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    >>> makeMatrix(4, 2)
    [[0, 0], [0, 0], [0, 0], [0, 0]]
    """
    return [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

UNIVERSITEIT GENT

Doctests

```
$ python matrix3.py
**********************************************************************
File "matrix3.py", line 8, in __main__.makeMatrix
Failed example:
    makeMatrix(4, 2)
Expected:
    [[0, 0], [0, 0], [0, 0], [0, 0]]
Got:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
**********************************************************************
1 items had failures:
   1 of   2 in __main__.makeMatrix
***Test Failed*** 1 failures.
$
```

UNIVERSITEIT GENT

Doctests

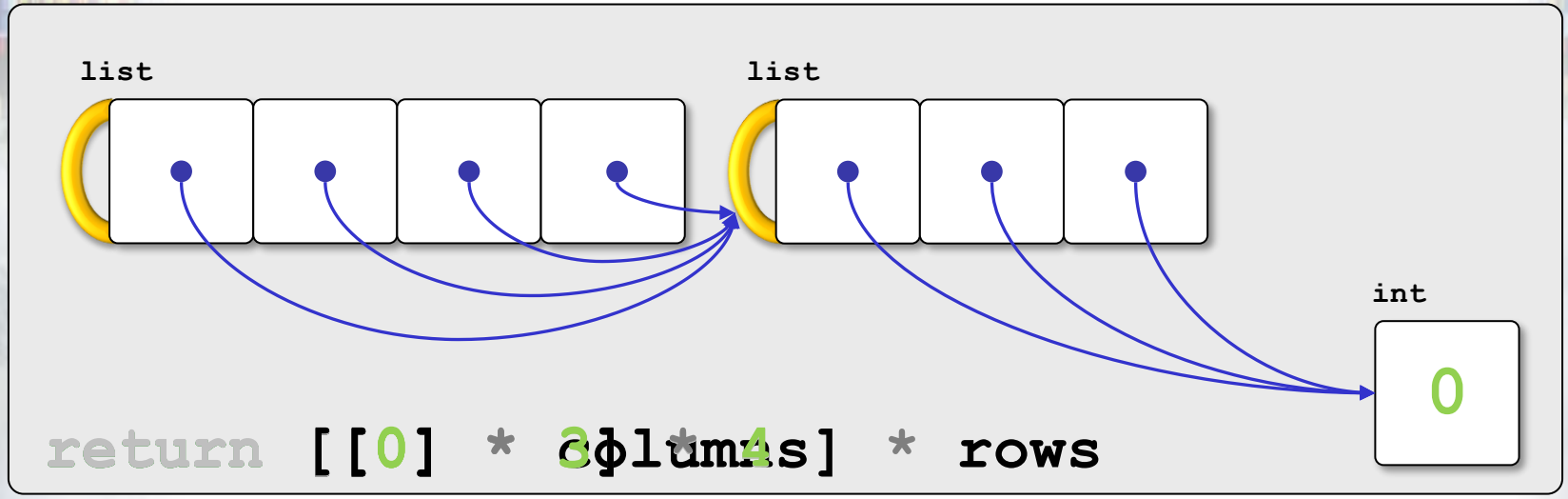
matrix4.py

```
def makeMatrix(rows, columns):
    """
    Returns a matrix with a given number of rows and
    columns. All matrix elements are initialized to 0.

    >>> makeMatrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    >>> makeMatrix(4, 2)
    [[0, 0], [0, 0], [0, 0], [0, 0]]
    """
    return [[0] * columns] * rows

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

```
$ python matrix4.py
$
```

UNIVERSITEIT GENT

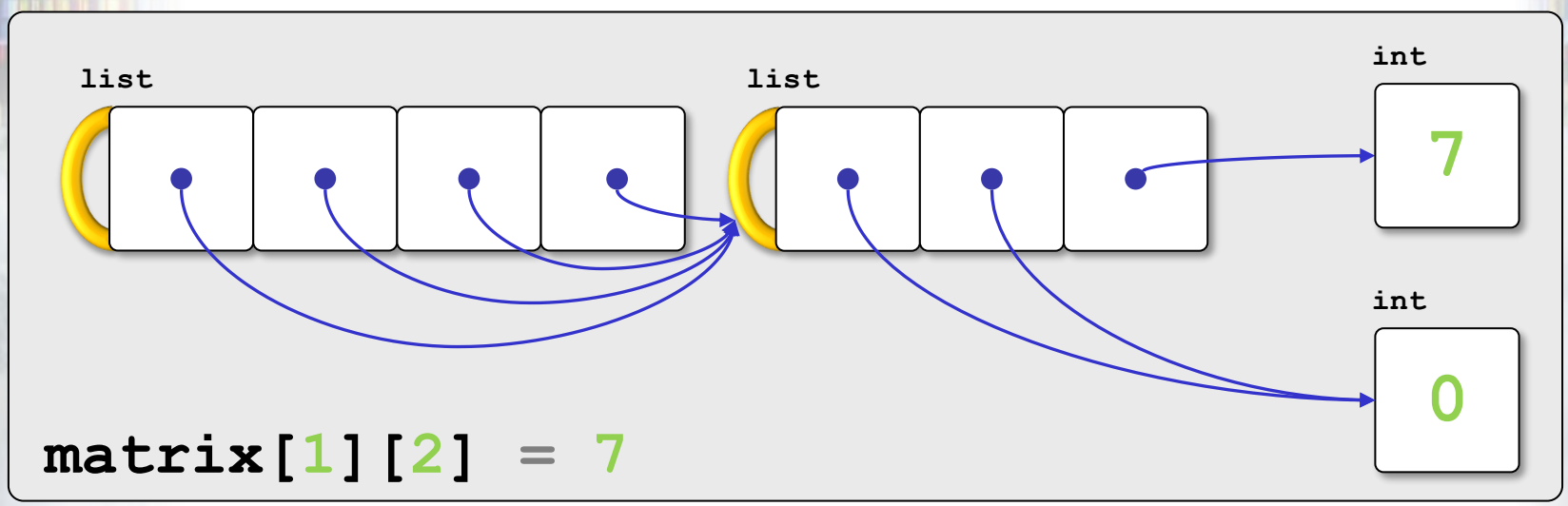
Doctests




```
>>> from matrix4 import *
>>> matrix = makeMatrix(4, 3)
>>> matrix
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> matrix[1][2] = 7
>>> matrix
[[0, 0, 7], [0, 0, 7], [0, 0, 7], [0, 0, 7]]
```

UNIVERSITEIT GENT

Doctests




```
>>> from matrix4 import *
>>> matrix = makeMatrix(4, 3)
>>> matrix
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> matrix[1][2] = 7
>>> matrix
[[0, 0, 7], [0, 0, 7], [0, 0, 7], [0, 0, 7]]
```

UNIVERSITEIT GENT

Doctests

matrix5.py

```
def makeMatrix(rows, columns):
    """
    Returns a matrix with a given number of rows and
    columns. All matrix elements are initialized to 0.

    >>> makeMatrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    >>> makeMatrix(4, 2)
    [[0, 0], [0, 0], [0, 0], [0, 0]]
    >>> matrix = makeMatrix(4, 2)
    >>> matrix[1][1] = 7
    >>> matrix
    [[0, 0], [0, 7], [0, 0], [0, 0]]
    """
    matrix = []
    for row in range(rows):
        matrix += [[0] * coloumns]
    return matrix

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

UNIVERSITEIT GENT

Homework (next lecture)

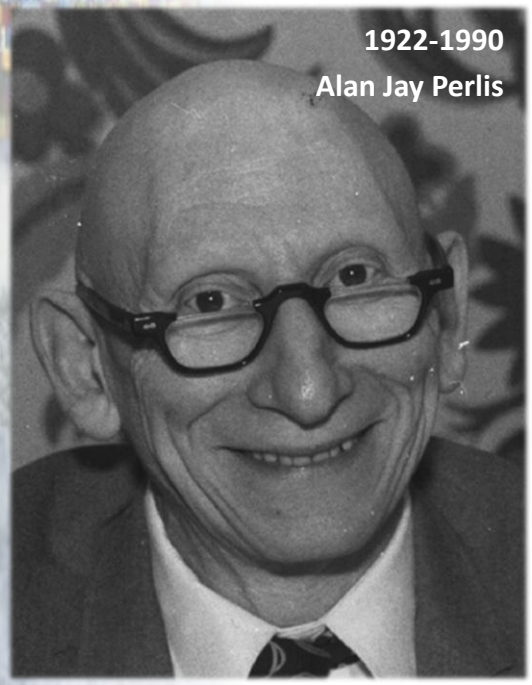
- course book
  - read chapter 9 (sets and dictionaries)
  - read chapter 10 (more program development)
- read problem description of classroom exercises
  - Simulating polymers
  - Cryptograms
  - Blood types

UNIVI
GENT

Questions or remarks?

UNIVERSITEIT GENT

The sky is the limit…



"If a listener nods his head when you're explaining your program, wake him up."

— Alan J. Perlis

UNIVERSITEIT GENT