Universidade Federal de Viçosa
Departamento de Informática
Centro de Ciências Exatas e Tecnológicas

Universidade Federal de Viçosa

INF 100 – Introduction to Programming

variables, operators,
input and output of data

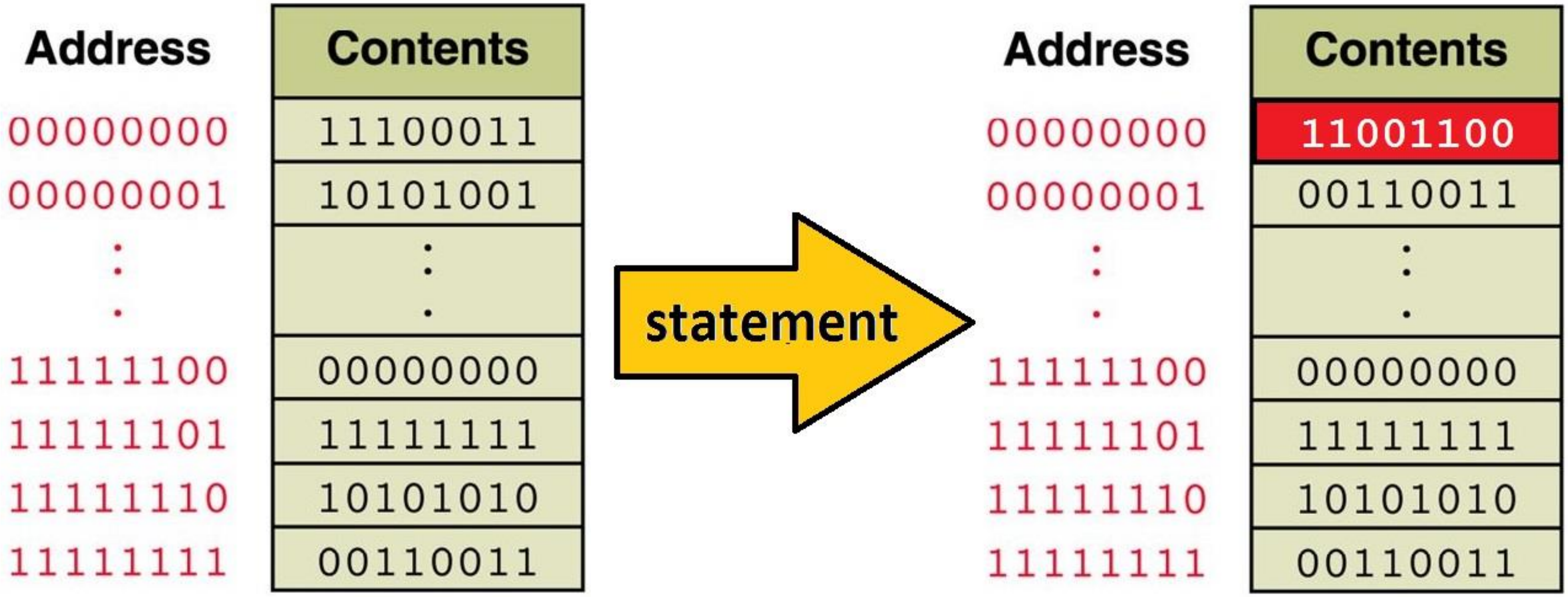
Imperative programming

- Several programming languages such as Python follow the paradigm[1] known as **imperative programming**.
- This paradigm describes computation in terms of **statements** that change the memory state (values for the memory cells).

[1] A **paradigm** is a distinct set of concepts or thought patterns; a model.

INF 100 – Introdução à Programação

Imperative programming



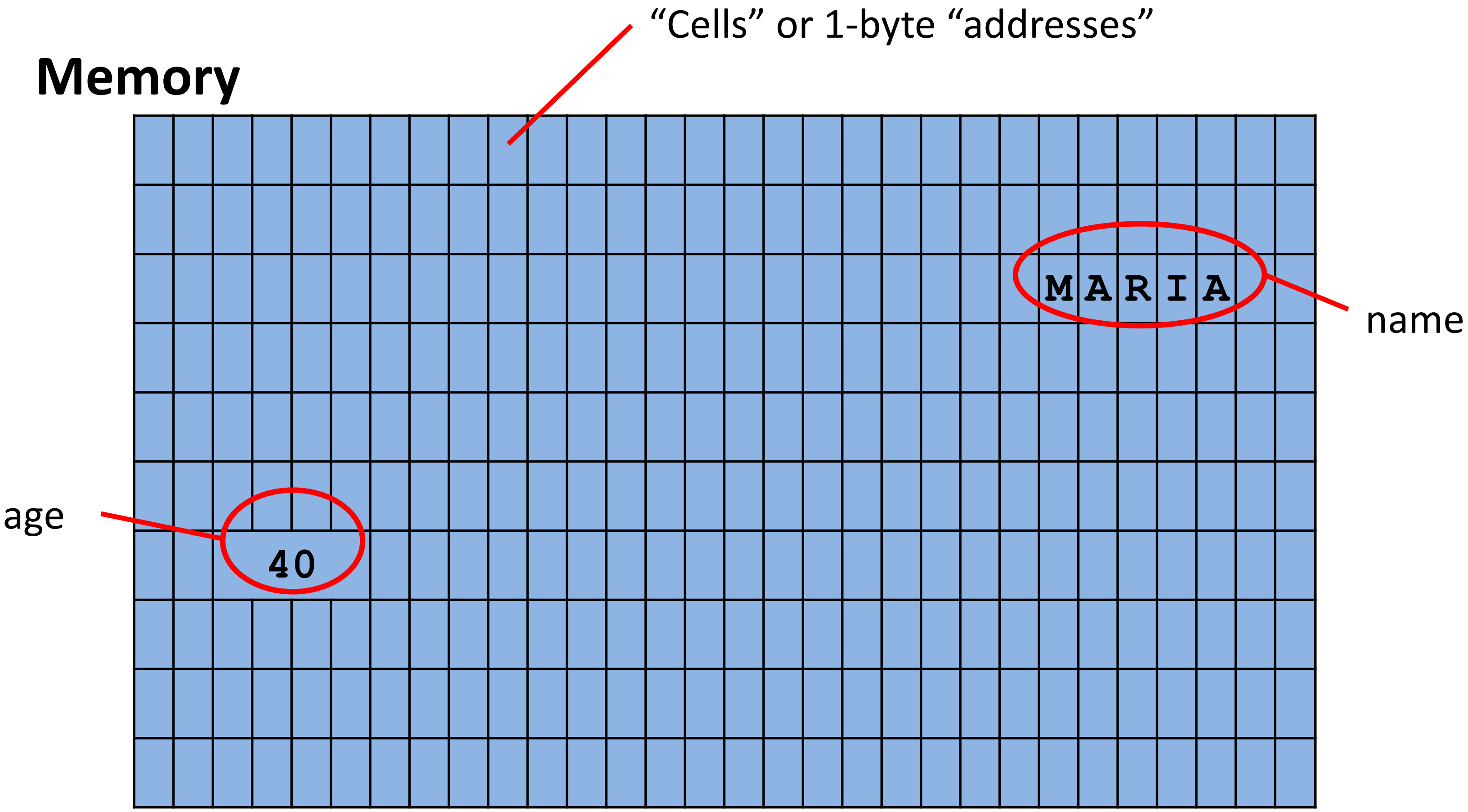
Address	Contents
00000000	11100011
00000001	10101001
⋮	⋮
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

Address	Contents
00000000	11001100
00000001	00110011
⋮	⋮
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

Variables

- In order to abstract from real memory positions, high level languages use the concept of **variables**.
- Variables are abstractions of a memory cell or a collection of memory cells.

Variables



Names for variables

- In Python, legal variable names must:
  - start with a letter or '_' (underline);
  - be followed by letters, '_' (underline) or digits.
- Examples of valid names:

  a *A* *abc* *x1* *var44* *_myVar* *another_var*
- Python is **case sensitive**

  so *abc* , *ABC* and *Abc* , for example, are considered as different variables.

Names for variables

- Some words are “reserved” by the language, to denote special values or commands. They cannot be used as variable names.
- In Python, the list of reseverd words can be checked with the following operations:

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as',
'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

Names for variables

- Examples of invalid names for variables:

```
8y
π
and
for
Large name
```

OBS: blank spaces are not allowed in variable names!

Names for variables

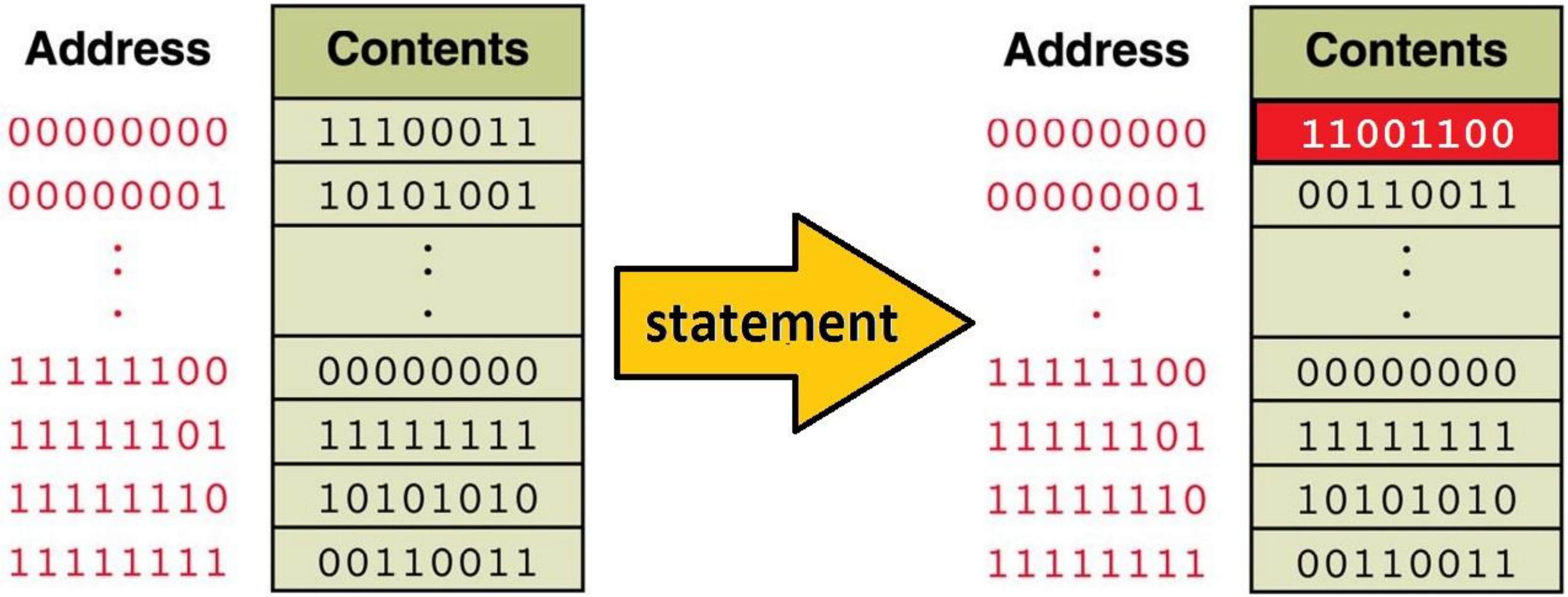
- Example: the mathematical formula

$$\Delta = \pi.(r_1 \; - r_2) \text{ . } \textit{impact factor}$$

could be represented in Python as

```
delta = PI*(r1-r2)*impact_factor
```

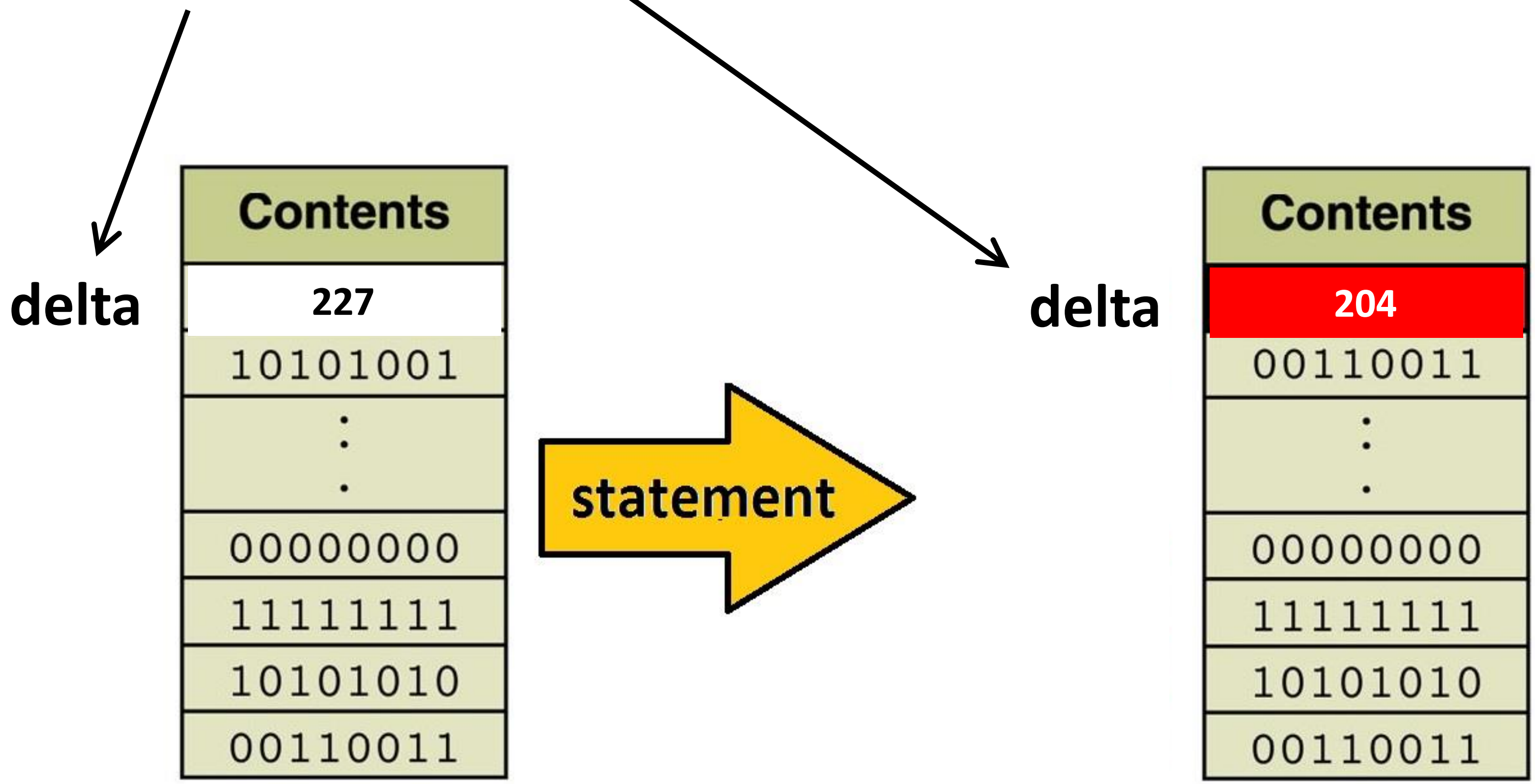
Using variables...



Address	Contents
00000000	11100011
00000001	10101001
:	:
.	.
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

Address	Contents
00000000	11001100
00000001	00110011
:	:
.	.
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

Using variables...



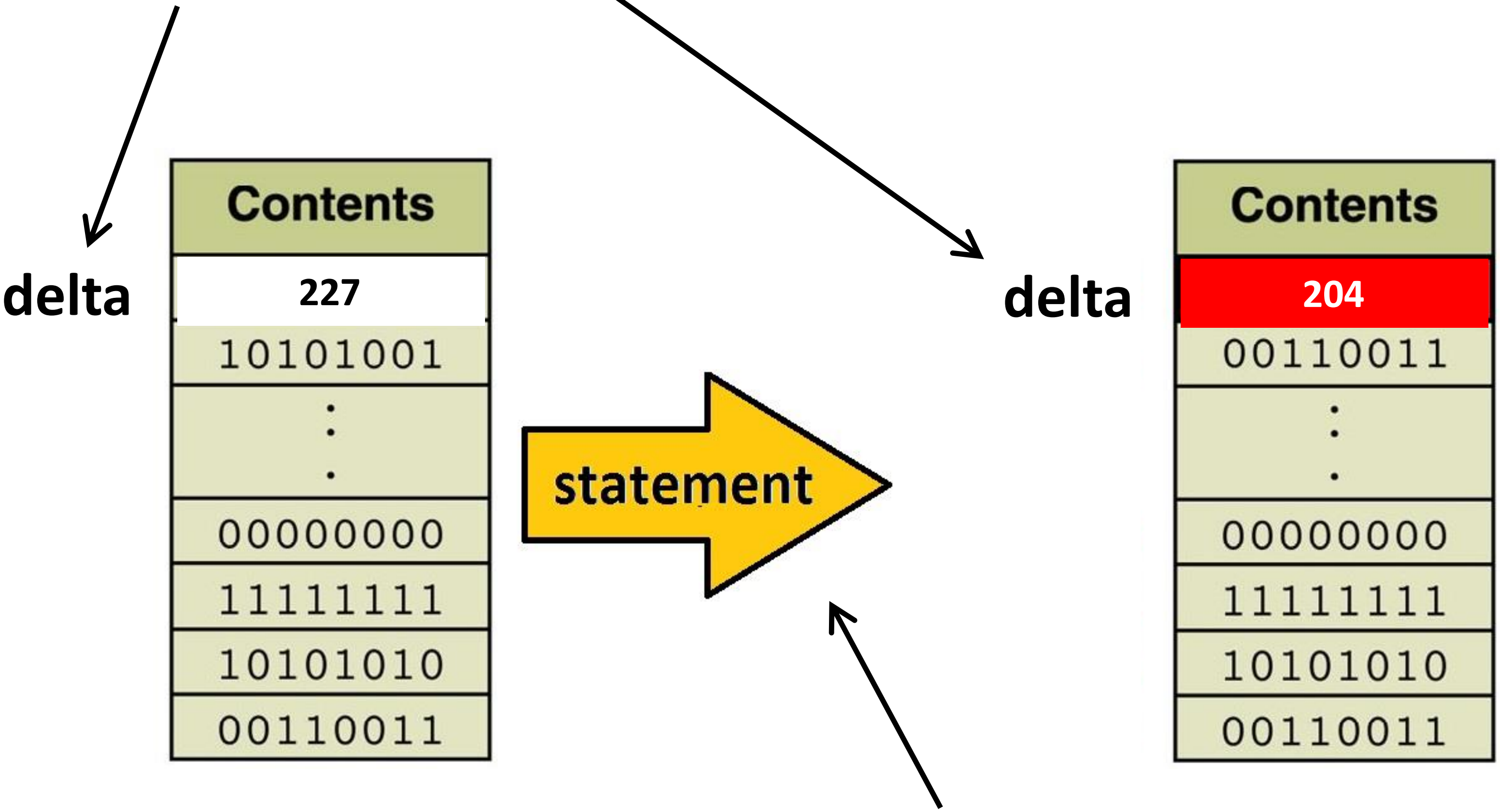
delta

Contents
227
10101001
⋮
00000000
11111111
10101010
00110011

delta

Contents
204
00110011
⋮
00000000
11111111
10101010
00110011

Using variables...



Let’s discuss statements now!

Statements

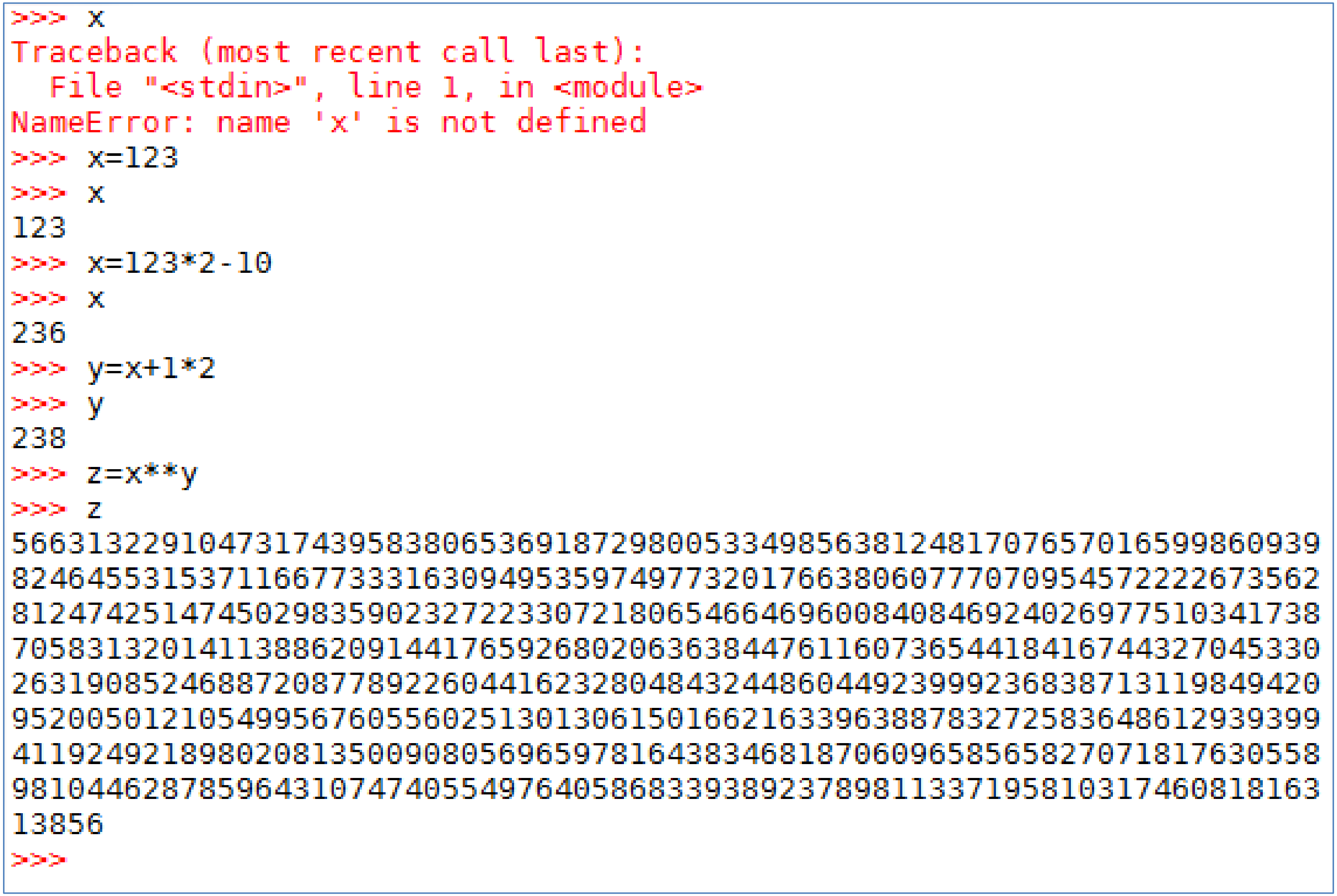
- The statements of a imperative programming language can read or modify variables.
- Some types of statements:
  - for reading data from input devices and storing it in variables;
  - for sending the value of variables to output devices;
  - for updating the value of a variable with the result of the evaluation of complex expressions that may involve several other variables.

Update statement

- In Python, an update statement has the following syntax (simplified):

nameVar = expression

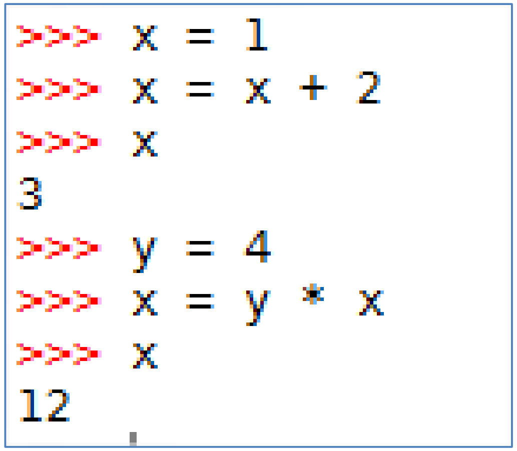
- Semantics: the expression on the right is calculated and then the result is stored in the variable whose name is specified on the left



```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x=123
>>> x
123
>>> x=123*2-10
>>> x
236
>>> y=x+1*2
>>> y
238
>>> z=x**y
>>> z
5663132291047317439583806536918729800533498563812481707657016599860939
8246455315371166773331630949535974977320176638060777070954572222673562
8124742514745029835902327223307218065466469600840846924026977510341738
7058313201411388620914417659268020636384476116073654418416744327045330
2631908524688720877892260441623280484324486044923999236838713119849420
9520050121054995676055602513013061501662163396388783272583648612939399
4119249218980208135009080569659781643834681870609658565827071817630558
9810446287859643107474055497640586833938923789811337195810317460818163
13856
>>>
```

Update statement

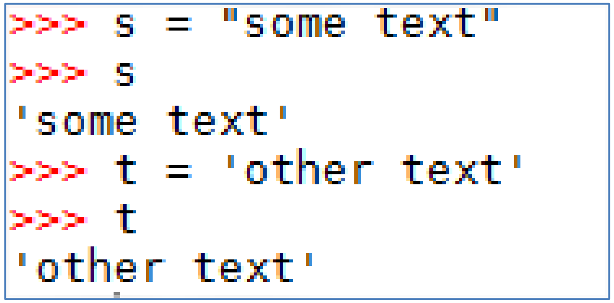
- An update statement can use the same variable name on the left and on the right side of the statement.



```
>>> x = 1
>>> x = x + 2
>>> x
3
>>> y = 4
>>> x = y * x
>>> x
12
```

Values

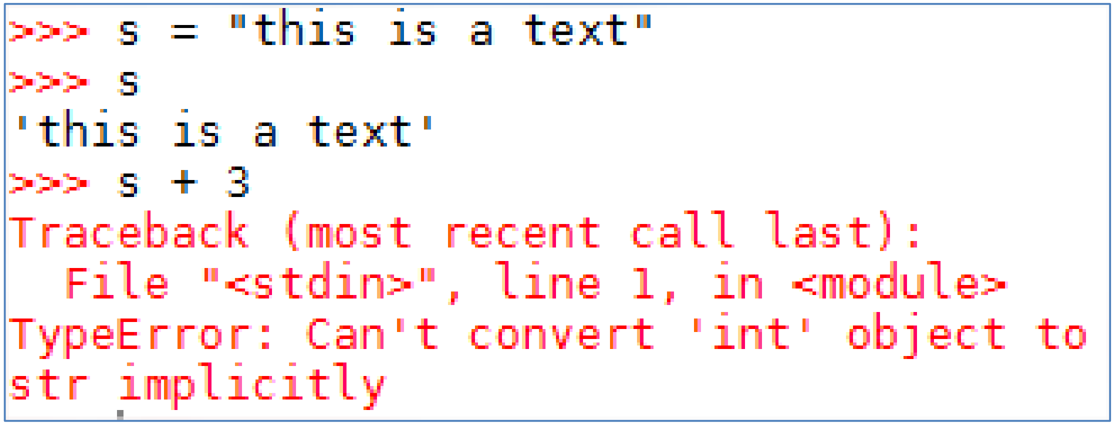
- Constant values can be integer constants, real constants, literal constants (texts)
- Literal constants are enclosed in quotation marks (either single or double quotes).



```
>>> s = "some text"
>>> s
'some text'
>>> t = 'other text'
>>> t
'other text'
```

Values

- Each value has an associated **type**
- To each type, there is a set of allowed operations



```
>>> s = "this is a text"
>>> s
'this is a text'
>>> s + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to
str implicitly
```

Arithmetic expressions

- Main operators in arithmetic expressions:

Operator	Name/Role
( )	Parentheses
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder (for integer division)
//	Quotient (for integer division)
**	Exponentation

Properties of operators

- **Arity**: number of operands (unary, binary, ternary...)
- **Return value**: the value returned by the execution of the operation.
- **Precedence**: an operator with higher precedence executes first.
- **Associativity**: defines order of execution when operators with same precedence are involved.

Arithmetic expressions

Operator	Precedence	Associativity	return value
( )	↑	-	-
**		right	exponentiation
* / // %		left	multiplication, quotient, remainder
+ -		left	sum, difference

Arithmetic expressions

- Examples:

Expression	Result
8+3*2	
(8+3)*2	
2+4%3	
(2+4)%3	
5-3+1	
5-(3+1)	
2**3+1	
2**(3+1)	
2**3**2	
(2**3)**2	

Arithmetic expressions

- Examples:

Expression	Result
`8+3*2`	14
`(8+3)*2`	22
`2+4%3`	3
`(2+4)%3`	0
`5-3+1`	3
`5-(3+1)`	1
`2**3+1`	9
`2**(3+1)`	16
`2**3**2`	512
`(2**3)**2`	64

Special assignment commands

- Perform an operation followed by an assignment.
- Advantage: simplify the code!

Expression	Equivalent to:
x += 2	x = x + 2
y -= 4	y = y - 4
z *= 2	z = z * 2
w /= 5	w = w / 5
t %= 10	t = t % 10

Print statement

- Syntax :
  - Word print, followed by an open parenthesis, followed by elements to be printed or special commands, followed by a close parenthesis.
  - The elements must be separated by commas.
- Semantics :
  - The elements are printed on the output device.

Print statement

```
# -*- coding: utf-8 -*-

age= 40
sex = "M"
name = "Carlos Alberto"
cable = "Pirelli's antichama"
width = '20"'
print( "Name:", name )
print( "Sex:", sex, "   Age:", age )
print( cable, width )
```

Print statement

```
# -*- coding: utf-8 -*-


age= 40
sex = "M"
name = "Carlos Alberto"
cable = "Pirelli's antichama"
width = '20"'
print( "Name:", name )
print( "Sex:", sex, "   Age:", age )
print( cable, width )
```

```
Name: Carlos Alberto
Sex: M    Age: 40
Pirelli's antichama 20"
```

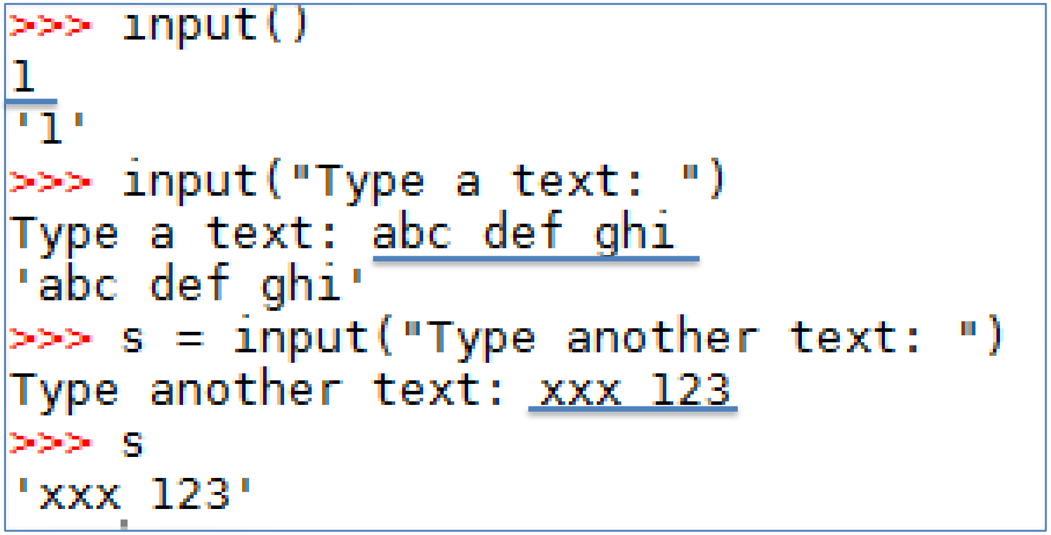
Basic input of data

- Command **input**: allow the program to read a text typed by the used on the keyboard, which is the standard input device.
- Syntax (simplified):

input (message_to_the_user)

Basic input

- Semantics: when a command **`input`** is interpreted, the program waits until the user types a text finishing with ENTER. The text is transformed into a text value that can be used inside the program.



```
>>> input()
1
'1'
>>> input("Type a text: ")
Type a text: abc def ghi
'abc def ghi'
>>> s = input("Type another text: ")
Type another text: xxx 123
>>> s
'xxx 123'
```

Basic input

- In order to work with input of numerical values, explicit conversion is necessary.
- Use int(...) to convert a text to an integer, and use float(...) to convert a value to a real number.

```
age = int (input("Type your age: "))
height = float (input("Type your height: "))
print("Age= ", age )
print("Height= ", height, "m")
```

Exercise

- Write a program in Python that reads 3 real values a, b e c from the keyboard and then prints the average of these values.

Use the technique of successive refinements.

First version

```
Algorithm:

read the values of a, b and c
calculate m as the average of a, b and c
print m
```

- Using incremental refinement, we start with a very simple version of an algorithm. We then refine each line into more precise commands, until we are able to produce a complete program that can be compiled.
- This technique is particularly interesting for very large or complex problems.

INF 100 – Introdução à Programação

First version

```
Algorithm:

read the values of a, b and c
calculate m as the average of a, b and c
print m
```

Attention:
The ORDER of these statements is very important!!

- Using incremental ref
version of an algorith
precise commands, u
complete program th
- This technique is particularly interesting for very large or complex problems.

Second version

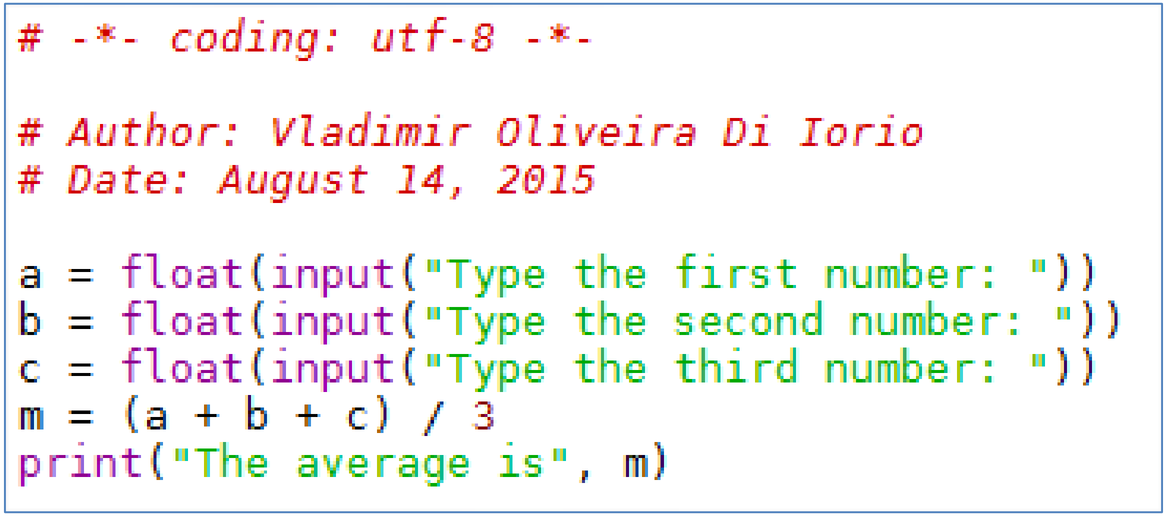
```
read a value and store in variable a
read a value and store in variable b
read a value and store in variable c
calculate m = (a + b + c) / 3
print "The average is " m
```

- The solution above is slightly more detailed than the first one.

- The benefits of refinements may not be very clear in this specific case, but it is an important tool when we will deal with more complex problems.

INF 100 – Introdução à Programação

Complete solution



```
# -*- coding: utf-8 -*-

# Author: Vladimir Oliveira Di Iorio
# Date: August 14, 2015

a = float(input("Type the first number: "))
b = float(input("Type the second number: "))
c = float(input("Type the third number: "))
m = (a + b + c) / 3
print("The average is", m)
```