Lesson 3a: Python Basics

History of Python language

- It was created by Guido Van Rossum (1991) in his free time when he worked on the Amoeba operating system team (distributed OS developed by A. Tanenbaum).
- Python was created to solve the problems Amoeba had when making system calls when connecting to Bourne Shell (interpreter[1] of default Unix version 7 systems).
- Python is inspired by languages ABC, Modula 3 and C.
- It's an interpreted language.

[1] Computer program that translates and executes the commands that users introduce.

Python properties

- Python supports several programming paradigms[2]:
  - **Object-oriented**: Provides a programming model based on objects that contain associated data and procedures known as methods.
  - **Imperative**: instructions are executed sequentially, one after another, unless conditional control structures are found.
  - **Functional**: treats computing as a process of application of functions, avoiding moving data with their changes of state.
- It is a **high-level language**. It manages its resources (especially memory). It is easy to read and flexible.
- It is a **strongly typified** language[3] and **dynamic typed**[4].

[2] A paradigm is a way of representing and manipulating knowledge.

[3] Any attempt to perform an operation on the wrong type triggers an error.

[4] It determines the type of operations at runtime.

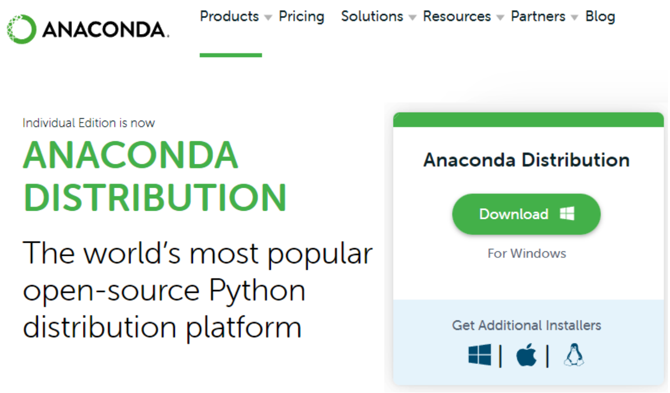
Advantages of Python

- It is a **very portable language**, that is, it is independent of the architecture of the machine. Python programs run on a virtual machine.
- Its grammar is elegant. It allows writing **clear, concise and simple code**, and develop a lot of use cases in an elegant and natural way.
- **Large set of functions** in the standard library and lots of external projects with specific functionalities.
- Its **interface** with **C, C++** and **Java** facilitates the connection between heterogeneous programs.

Disadvantages of Python

- Since it is an interpreted language, it is considered a **slow language**, although Python 3 improves its performance.
- It is **less extended than other languages** such as C/C++ or Java, although it is increasing.
- It is **not a very good** language **for mobile development**.
- It's **not a good choice for memory-intensive** tasks.
- It has **limitations** regarding the access to **databases**.

Python on operating systems

<table>
<tr><th>Windows</th><th>MacOS</th><th>Linux</th></tr>
<tr><td>Not installed</td><td>Usually pre-installed</td><td>Usually pre-installed</td></tr>
<tr><td colspan="3">An editor to write programs is needed</td></tr>
<tr><td colspan="3">We recommend installing the Anaconda distribution


ANACONDA.
Products
Pricing
Solutions
Resources
Partners
Blog
Anaconda Distribution
Download
For Windows
Get Additional Installers

Individual Edition is now
ANACONDA DISTRIBUTION
The world's most popular open-source Python distribution platform
https://www.anaconda.com/products/individual</td></tr>
</table>

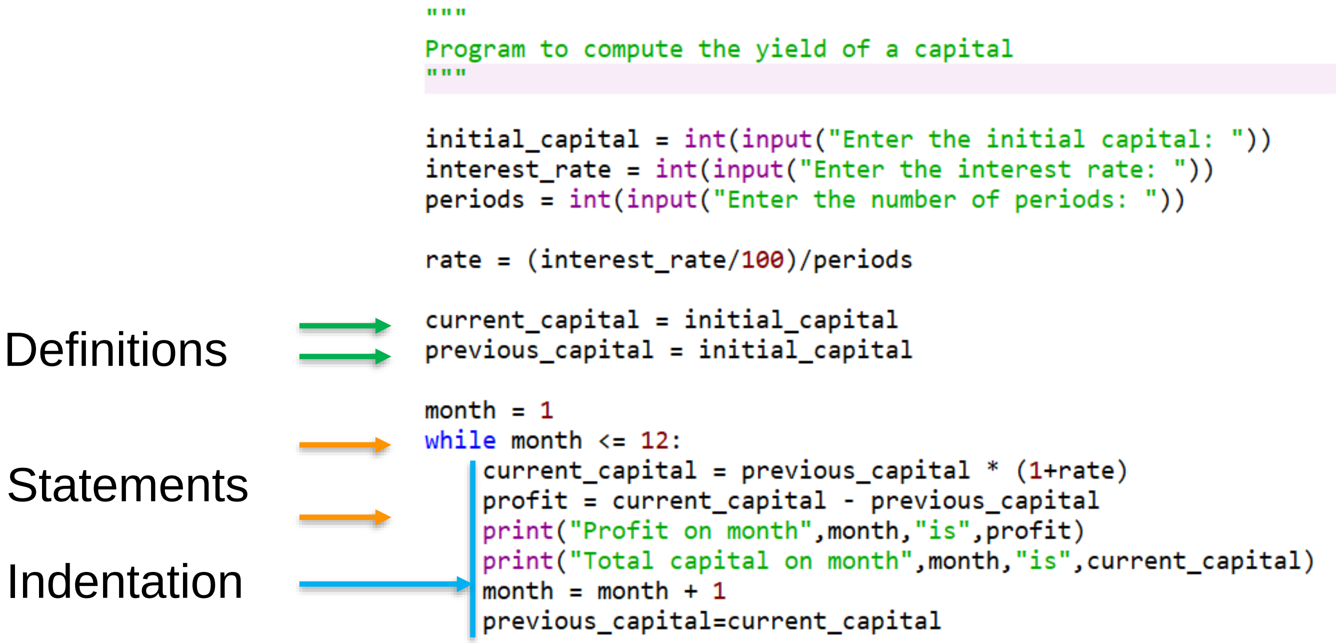
The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one, and preferably only one, obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never if often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Basic elements in Python

- A program in Python is a sequence of definitions and statements:
  - Definitions are evaluated
  - Statements are executed by Python Interpreter (Shell)

- Statements are instructions to do something.



Definitions

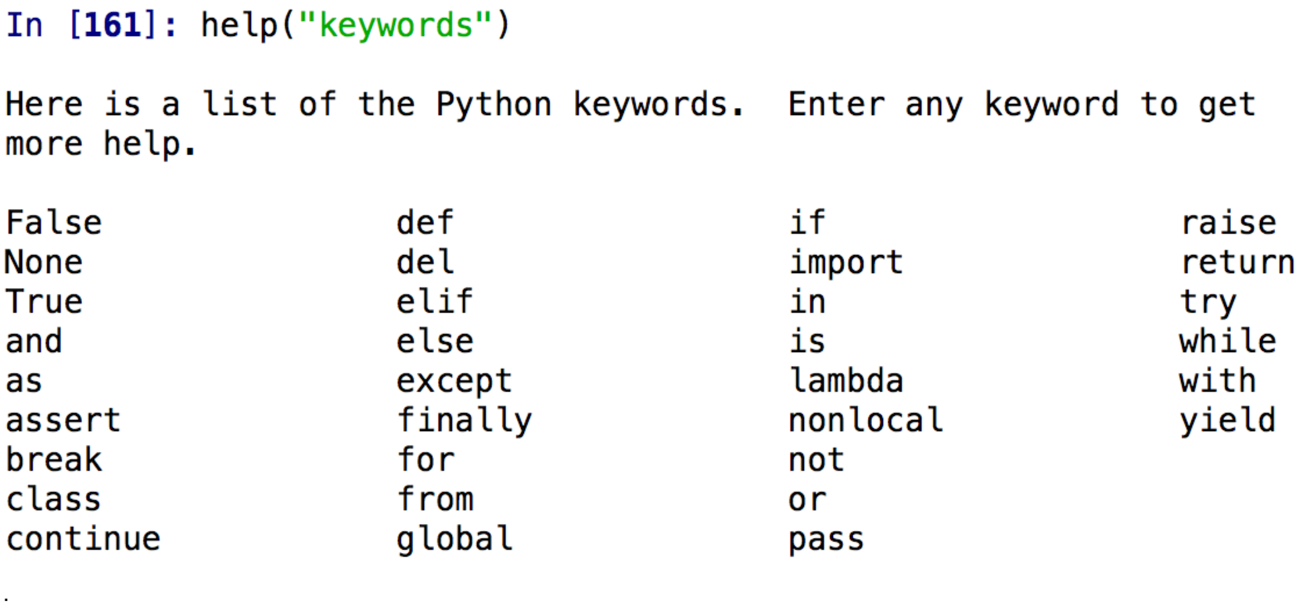
Statements

Indentation

```
"""
Program to compute the yield of a capital
"""

initial_capital = int(input("Enter the initial capital: "))
interest_rate = int(input("Enter the interest rate: "))
periods = int(input("Enter the number of periods: "))

rate = (interest_rate/100)/periods

current_capital = initial_capital
previous_capital = initial_capital

month = 1
while month <= 12:
    current_capital = previous_capital * (1+rate)
    profit = current_capital - previous_capital
    print("Profit on month",month,"is",profit)
    print("Total capital on month",month,"is",current_capital)
    month = month + 1
    previous_capital=current_capital
```

Synth elements of a program in Python

Reserved words: These are the basic instructions of Python



```
In [161]: help("keywords")

Here is a list of the Python keywords.  Enter any keyword to get
more help.

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

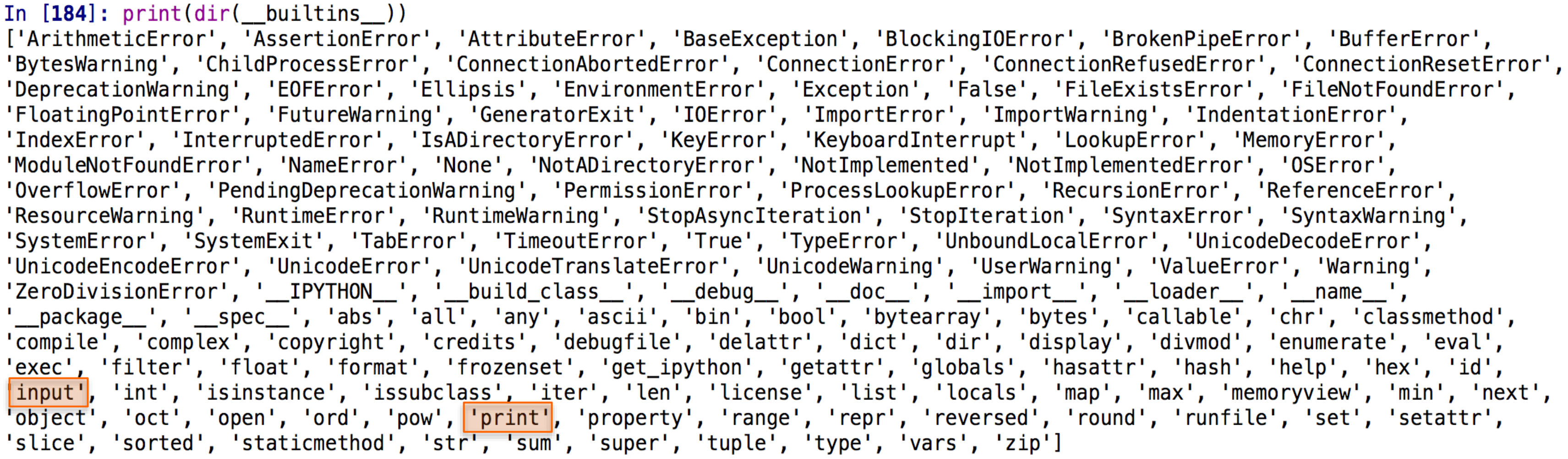
... and the instructions for printing on the screen and reading from the keyboard ?

Advantages of Python

- It is a **very portable language**, that is, it is independent of the architecture of the machine. Python programs run on a virtual machine.
- Its grammar is elegant. It allows writing **clear, concise and simple code**, and develop a lot of use cases in an elegant and natural way.
- **Large set of functions** in the standard library and lots of external projects with specific functionalities.
- Its **interface** with **C, C++** and **Java** facilitates the connection between heterogeneous programs.

Synth elements of a program in Python

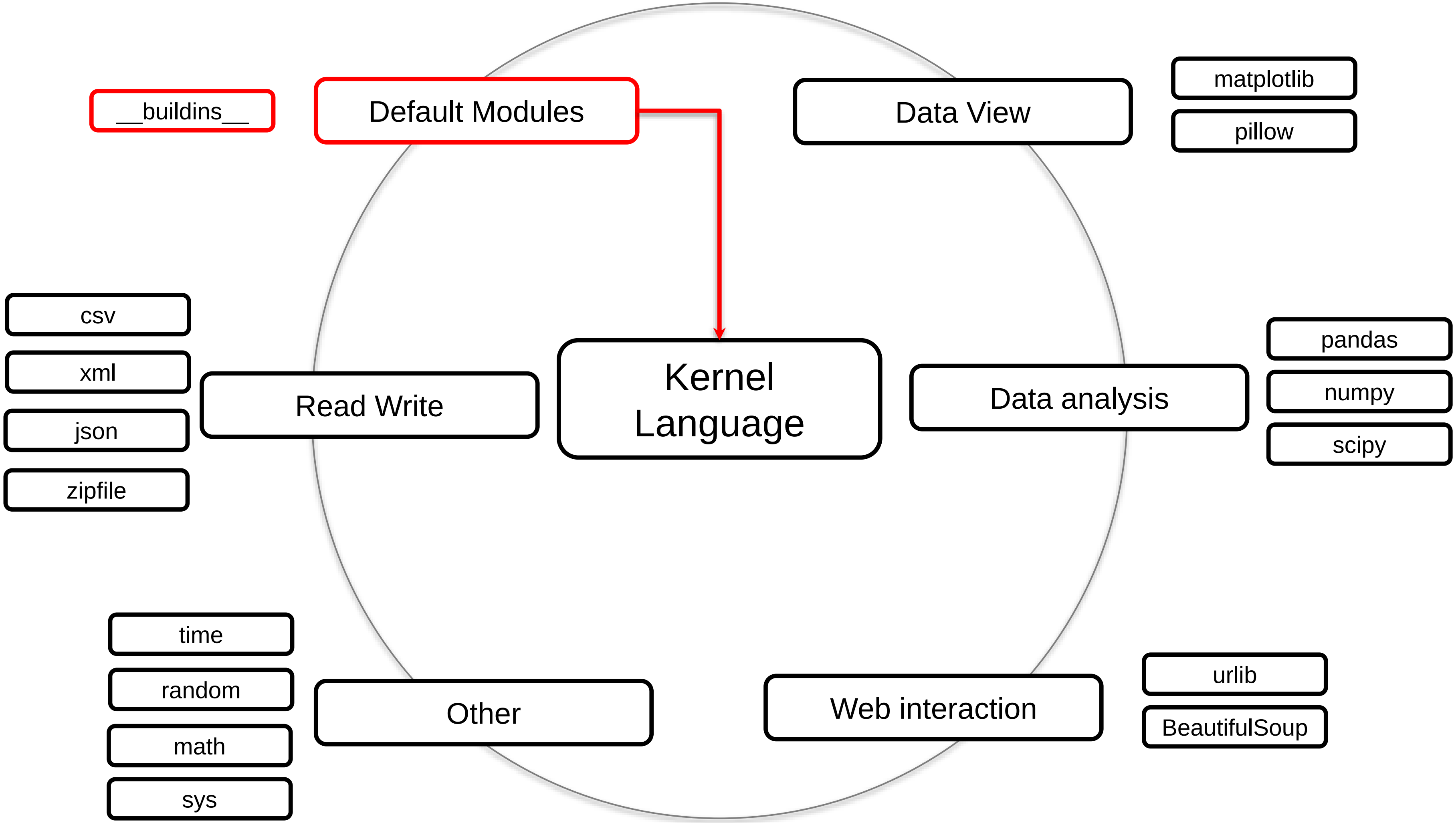
They are not part of the kernel, but they are a fundamental part of language:



```
In [184]: print(dir(__builtins__))
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__',
'__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',
'compile', 'complex', 'copyright', 'credits', 'debugfile', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval',
'exec', 'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'runfile', 'set', 'setattr',
'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

instructions for writing and reading... and some more

Modules


__buildins__
Default Modules
Data View
matplotlib
pillow
csv
xml
json
zipfile
Read Write
Kernel Language
Data analysis
pandas
numpy
scipy
time
random
math
sys
Other
Web interaction
urlib
BeautifulSoup

Variables

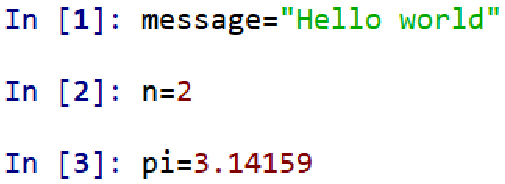
All **values are stored in the computer memory**.

We can imagine the memory as a shelf where boxes contain data.

A variable is the name to refer to and to access a value/object.

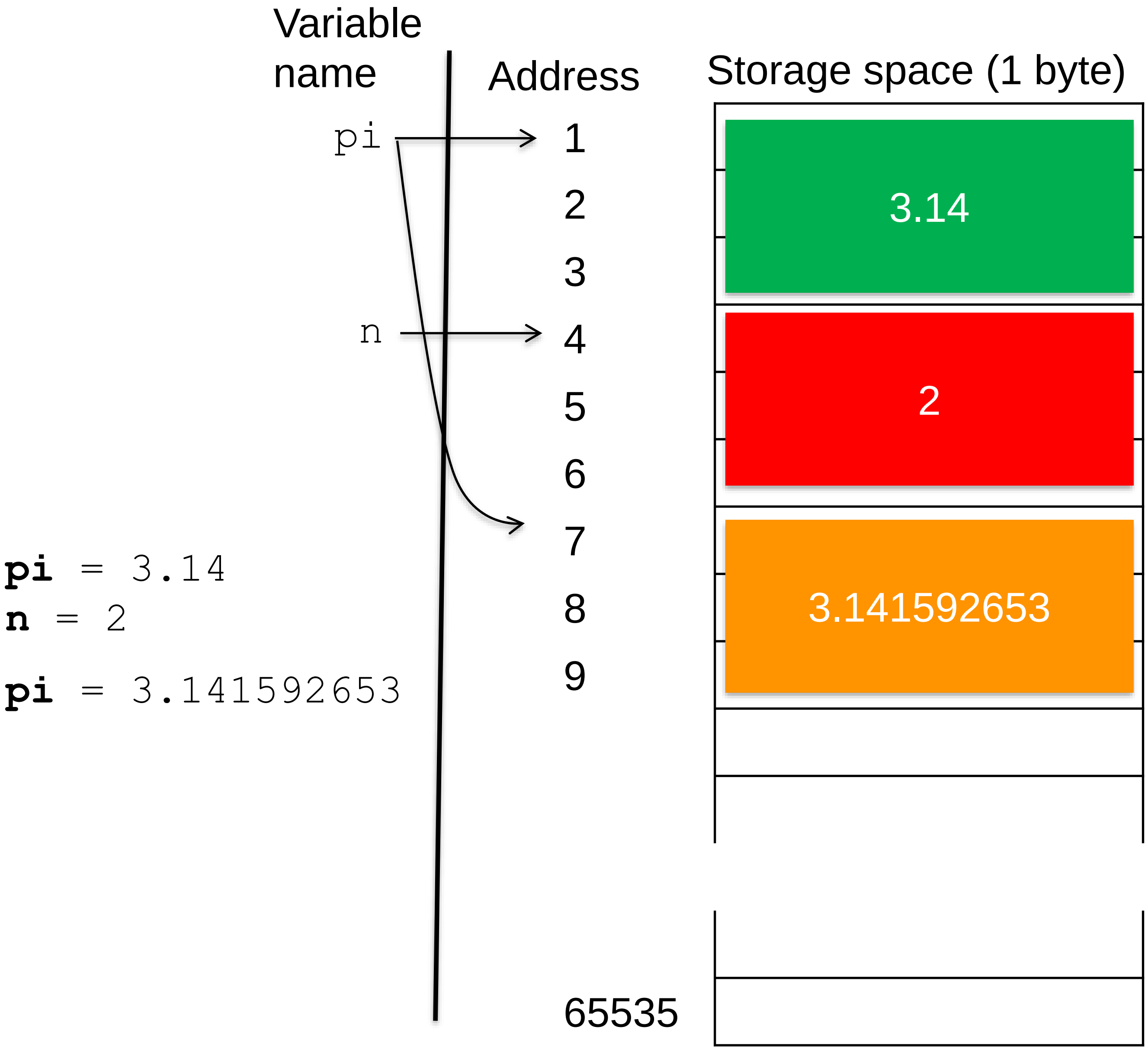
In Python, unlike in other languages, **variables must NOT be declared** previously to be used.

When we make an assignment (=), we are creating a variable (the one to the left of the =) and giving it a value (the one to the right of the =)



```
In [1]: message="Hello world"

In [2]: n=2

In [3]: pi=3.14159
```

Memory organization



```
pi = 3.14
n = 2

pi = 3.141592653
```

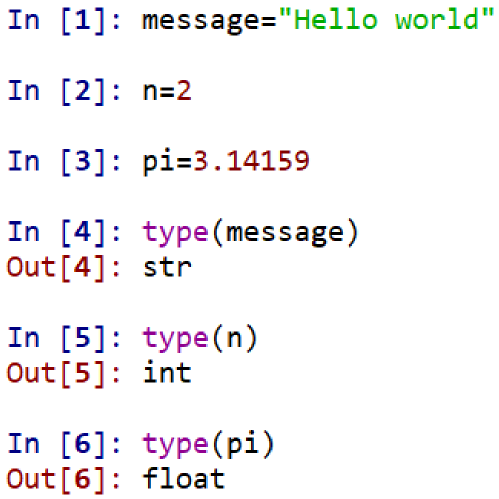
1 byte = 8 bits

So many positions as memory has the computer

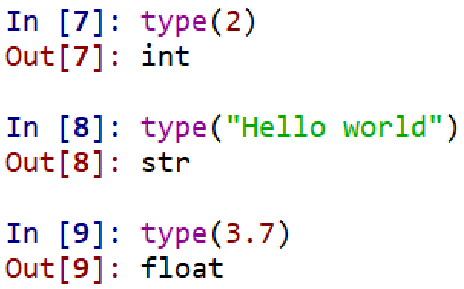
Types

- Each variable has a type depending on the data it stores.

- Basic types of data in Python:
  - `int` is used to represent integers (e.g. `-1` , `4` , `2300`)
  - `float` is used to represent real numbers (e.g. `-1.1` , `4.3` , `2.3E3`)
  - `bool` is used to represent Boolean values (`True` and `False`)
  - `str` is used to represent a set of characters (e.g. "`Hello`" , "`A23`")



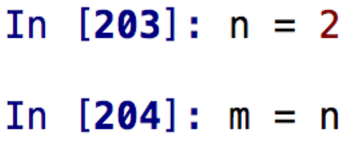
```
In [1]: message="Hello world"

In [2]: n=2

In [3]: pi=3.14159

In [4]: type(message)
Out[4]: str

In [5]: type(n)
Out[5]: int

In [6]: type(pi)
Out[6]: float
```



```
In [7]: type(2)
Out[7]: int

In [8]: type("Hello world")
Out[8]: str

In [9]: type(3.7)
Out[9]: float
```

Names of variables

- In Python, a variable is just a name.
- An assignment statement associates the name on the left of = with the object denoted by the expression on the right of =.
- A value can have one, more than one or no names associated with it:



```
In [203]: n = 2

In [204]: m = n
```

- But names are important:

```
a = 3.14159             pi = 3.14159
b = 11.2                radius = 11.2
c = a*(b**2)            area = pi*(radius**2)
```

Both programs are exactly the same for Python, but not for us...

Names of variables

Rules:

- Valid characters: letters (a.. z and A.. Z), digits (0-9) and underscore (_).
- Any other character is not allowed.
- First character: a letter or underscore. Cannot be a digit.
- Python's reserved words must not be used.
- Case sensitive:

area ≠ Area ≠ AREA

Examples:

Valid identifiers	**Invalid identifiers**
circle_area	1x
_circle_area	Circle Area
CircleArea_1	Circle#Area
x1	X:
ThisIdentifierIsValid	ThisIsNotV@lid

Constants

Constants: In many languages, values that do not change during execution are defined as constants.

In Phyton there is no difference between constants and variables.

It is a good programming practice to differentiate them. Usually, variables that do not change value in the program will be defined in uppercase:

```
PI = 3.141592
```

Comments

- A good practice to facilitate understanding the code is to add comments.
- Text after character # is not interpreted by Phyton:

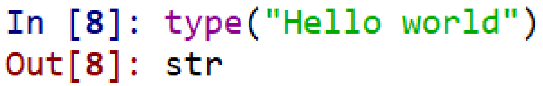
```
#Computes the area of a circle
PI = 3.14159
radius = 11.2
area = PI*radius**2
```

- To comment on more than one line, we use 3 double quotes in a row to indicate the beginning and the end of a comment:

```
"""
Computes the area of a circle
Version 27-12-2020
"""
PI = 3.14159
radius = 11.2
area = PI*radius**2
```

Function calls

- We have already seen some examples of a call to a function:



```
In [8]: type("Hello world")
Out[8]: str
```

- The function name is type, and it displays the type of a value or variable.

- The value or variable, which is called the function **argument** or **parameter**, must be included in parentheses.

- It is common to say that a function **"takes" an argument** and **"returns" a result**.

- The result is called **return value**.

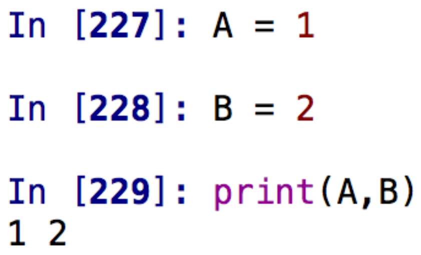
Data output

The standard output is usually the screen.

```
print ("Hello, World")
```

`print()` within the parenthesis we put what we want to print on the screen.

To print more than one item, we separate items with commas (,):



```
In [227]: A = 1

In [228]: B = 2

In [229]: print(A,B)
1 2
```

`print()` leaves a space between the elements

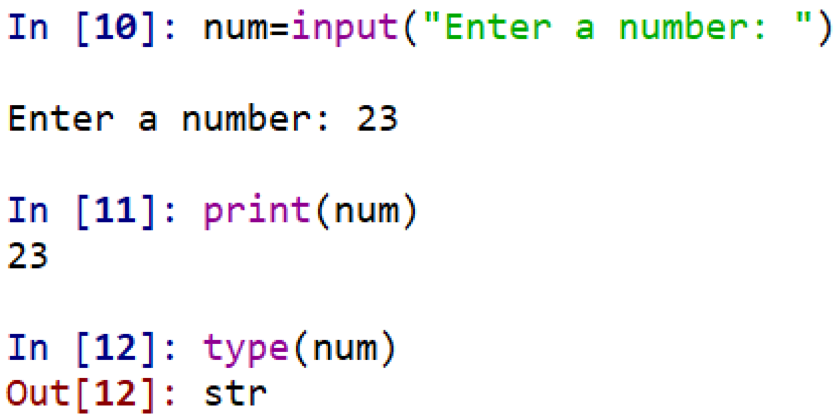
Data input

The standard input is usually the keyboard.

`input()` is a function that returns a value (the one read from the keyboard). Therefore, we must assign it to a variable:

```
num=input("Enter a number: ")
```

`input()` always returns a string. Therefore, input is of type `str` despite entering a number:

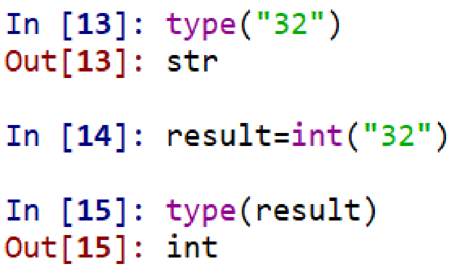


```
In [10]: num=input("Enter a number: ")

Enter a number: 23

In [11]: print(num)
23

In [12]: type(num)
Out[12]: str
```

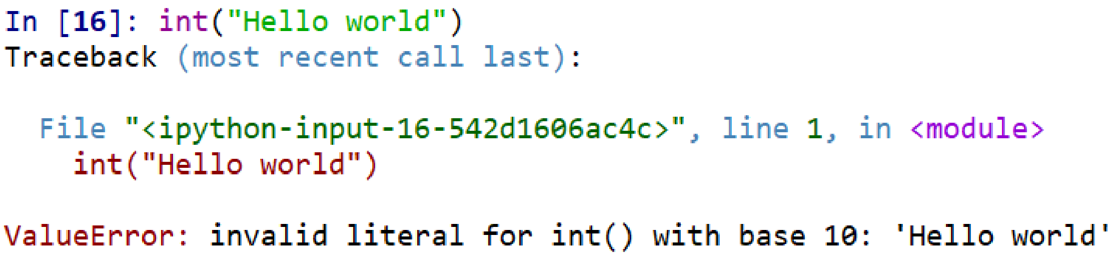
We will see later how we solve this.

Type conversion

- Function `int` takes any value and converts it to an integer if possible or gives an error.
- `int` can also convert floating point values to integers, but it truncates the fractional part:

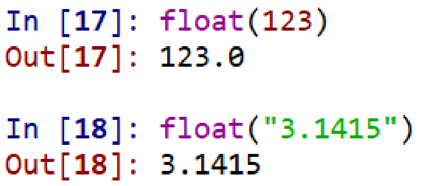


```
In [13]: type("32")
Out[13]: str

In [14]: result=int("32")

In [15]: type(result)
Out[15]: int
```

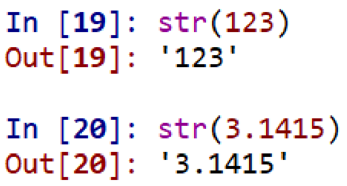


```
In [16]: int("Hello world")
Traceback (most recent call last):

  File "<ipython-input-16-542d1606ac4c>", line 1, in <module>
    int("Hello world")

ValueError: invalid literal for int() with base 10: 'Hello world'
```

- Function `float` converts integers and strings to float numbers
- Function `str` converts a value or variable into a string



```
In [17]: float(123)
Out[17]: 123.0

In [18]: float("3.1415")
Out[18]: 3.1415
```



```
In [19]: str(123)
Out[19]: '123'

In [20]: str(3.1415)
Out[20]: '3.1415'
```

Expressions

An expression is essentially a formula that allows calculating a value.

An expression is a combination of values, variables, and operators. If we type an expression on the command line, the interpreter evaluates it and displays the result.

We do not need to have all the elements to have an expression; a single value is also considered an expression

Any expression has a type. For example, the expression:

```
3.0 + 1.5
```

has type `float`.

The result of an expression can be assigned to a variable:

```
res = 3.0 + 1.5
```

Operators and operands

- Operators are special symbols that represent calculations such as addition, multiplication, etc.
- The values used by the operator are called operands

Arithmetic operators

Operation	Symbol	Type of operands*	Examples
Addition	+	int, float	5 + 3 → 8
Subtraction	–	int, float	5 – 3 → 2
Product	*	int, float	5 * 3 → 15
Division	/	int, float	10.0 / 3.0 → 3.333
Integer division	/ /	int, float	10 / /3 → 3
Modulus	%	int, float	10 % 3 → 1
Power	* *	int, float	10**2 →100

* If the two operands are int, the result is int (except for division, which returns float). If any of them is a float, the result is float.

Assignment operators

- Assignment operators are used to link a value to a name (variable).
- They allow compacting the instructions and writing less.

Assignment operators

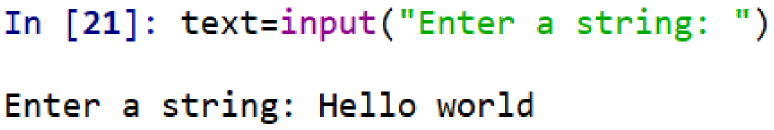
Operation	Symbol	Example
Addition	+=	x += y is equivalent to x = x + y
Subtraction	−=	x −= y is equivalent to x = x − y
Product	*=	x *= y is equivalent to x = x * y
Division	/=	x /= y is equivalent to x = x / y
Integer division	//=	x //= y is equivalent to x = x // y
Modulus	%=	x %= y is equivalent to x = x % y
Power	**=	x **= y is equivalent to x = x ** y

Precedence of operators

- When there is more than one operator in the expression, the evaluation order depends on the rules of precedence.
- Python follows the same rules of precedence for its mathematical operators as those used in mathematics:
  - *Parentheses* have the highest priority and can be used to force one's evaluation in the order we want.
  - The *exponent* is next in precedence.
  - *Multiplication* and *division* have the same priority
  - Next in precedence, we have *addition* and *subtraction*, with the same priority.
  - Operators with the same priority are evaluated from left to right.

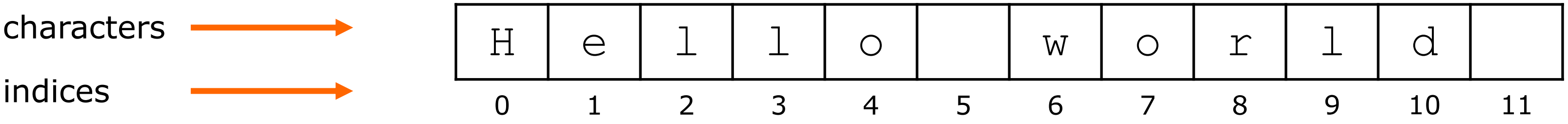
String Type (str)

- A string is a sequence of characters.



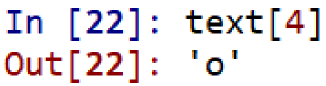
```
In [21]: text=input("Enter a string: ")

Enter a string: Hello world
```

- How is it stored?



characters	H	e	l	l	o		w	o	r	l	d	
indices	0	1	2	3	4	5	6	7	8	9	10	11

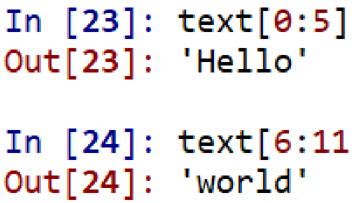
- Operator [ ] : access to a position



```
In [22]: text[4]
Out[22]: 'o'
```

e.g. 5th character (index 4)

- Operator [n:m] returns the part of the string from the n-th character (included) to the m-th (not included):



```
In [23]: text[0:5]
Out[23]: 'Hello'

In [24]: text[6:11]
Out[24]: 'world'
```

ASCII Code

The ASCII code is a Latin-based character code. It was created in 1963 by the American Committee on Standards (source: *Wikipedia*).

Essentially, it establishes a correspondence between the 256 integers that fit in a byte (8 bits of information = 2^8 possible combinations) and the alphabet.

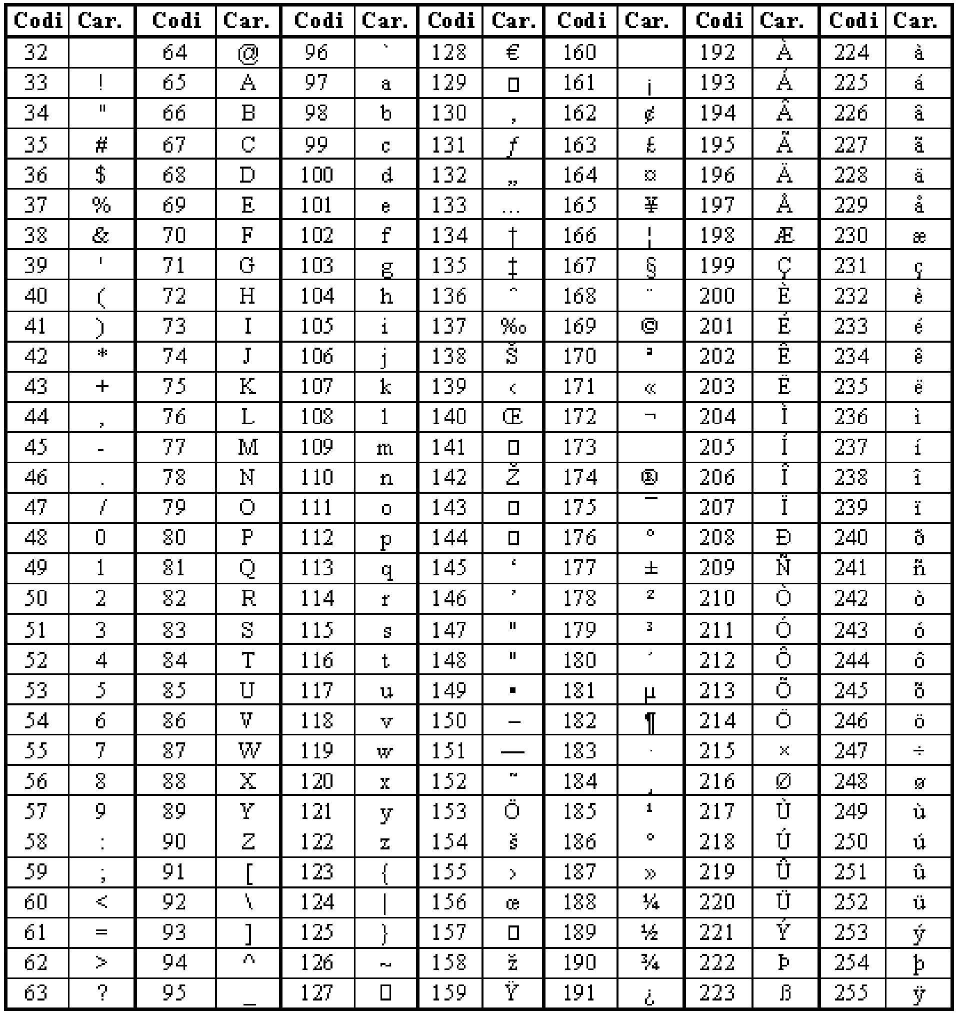
The most interesting values for us are the letters "a" and "A", and the digit "0". These characters have codes 97, 65 and 48 respectively.

The interest in choosing these comes from the fact that the entire lowercase alphabet comes after letter "a", so "b" has code 98, "c" has code 99, and so on until "z", which has code 122.

The same happens for capital letters: "B" is 66, "C" on 67, and so on until "Z", which is 90.

Numbers from "0" to "9" have codes from 48 to 57.

ASCII Code



Codi	Car.	Codi	Car.	Codi	Car.	Codi	Car.	Codi	Car.	Codi	Car.	Codi	Car.
32		64	@	96	`	128	€	160		192	À	224	à
33	!	65	A	97	a	129	□	161	¡	193	Á	225	á
34	"	66	B	98	b	130	‚	162	¢	194	Â	226	â
35	#	67	C	99	c	131	ƒ	163	£	195	Ã	227	ã
36	$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
37	%	69	E	101	e	133	…	165	¥	197	Å	229	å
38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
40	(	72	H	104	h	136	ˆ	168	¨	200	È	232	è
41	)	73	I	105	i	137	‰	169	©	201	É	233	é
42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
43	+	75	K	107	k	139	‹	171	«	203	Ë	235	ë
44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
45	-	77	M	109	m	141	□	173		205	Í	237	í
46	.	78	N	110	n	142	Ž	174	®	206	Î	238	î
47	/	79	O	111	o	143	□	175	¯	207	Ï	239	ï
48	0	80	P	112	p	144	□	176	°	208	Ð	240	ð
49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
51	3	83	S	115	s	147	"	179	³	211	Ó	243	ó
52	4	84	T	116	t	148	"	180	´	212	Ô	244	ô
53	5	85	U	117	u	149	▪	181	µ	213	Õ	245	õ
54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
55	7	87	W	119	w	151	—	183	·	215	×	247	÷
56	8	88	X	120	x	152	˜	184	¸	216	Ø	248	ø
57	9	89	Y	121	y	153	Ö	185	¹	217	Ù	249	ù
58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
59	;	91	[	123	{	155	›	187	»	219	Û	251	û
60	<	92	\	124	|	156	œ	188	¼	220	Ü	252	ü
61	=	93	]	125	}	157	□	189	½	221	Ý	253	ý
62	>	94	^	126	~	158	ž	190	¾	222	Þ	254	þ
63	?	95	_	127	□	159	Ÿ	191	¿	223	ß	255	ÿ

Each character corresponds to a number between 0 and 255.

On the keyboard:
ALT + Num → character from table

Example: ALT + 169 → ©

Python

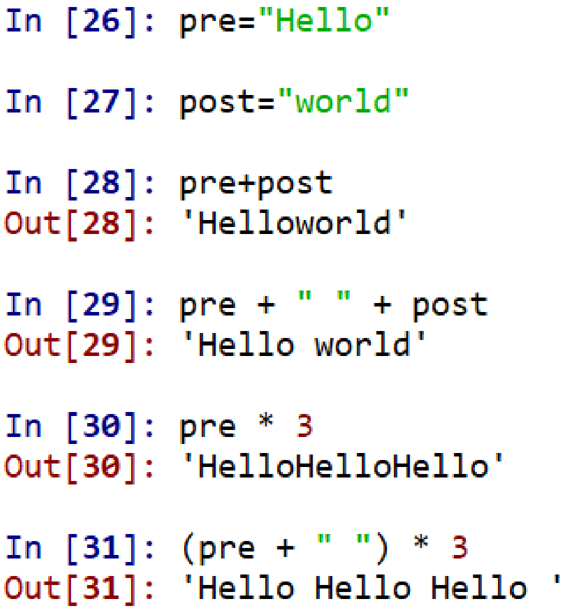
```
print(chr(65))    → 'A'
print(ord("A"))   → 65
```

Operators and strings

- In mathematics it makes no sense to operate with strings, although sometimes the strings look like numbers (e.g. "17"):

```
"17" + 1
```

- If we open our minds:
  - what could it mean to add two strings? → Concatenate them
  - what could it mean to multiply a string by an int? → Repeat it



```
In [26]: pre="Hello"

In [27]: post="world"

In [28]: pre+post
Out[28]: 'Helloworld'

In [29]: pre + " " + post
Out[29]: 'Hello world'

In [30]: pre * 3
Out[30]: 'HelloHelloHello'

In [31]: (pre + " ") * 3
Out[31]: 'Hello Hello Hello '
```

Which mathematical property does not have the sum of strings?

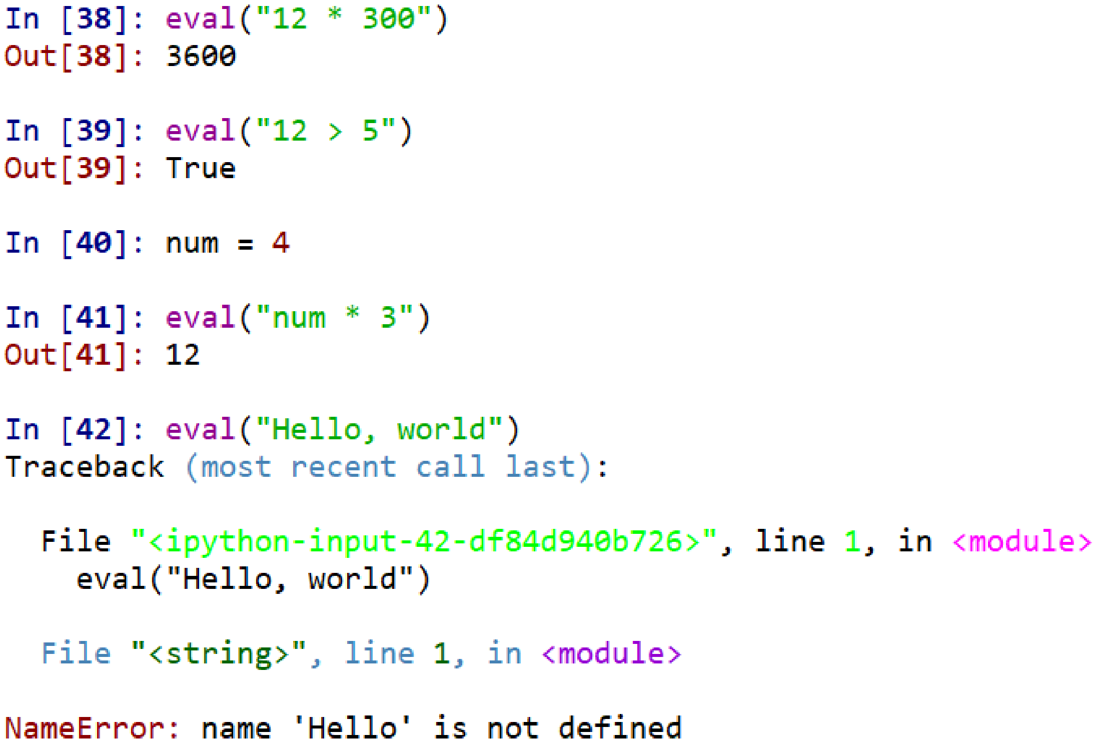
Commutative Property (a+b ≠ b+a)

Eval() function

- The `eval()` function evaluates a `str` object as if it was an expression:

```
eval (<str object>)
```

- If the text to evaluate is not a valid expression, eval() will generate an error message



```
In [38]: eval("12 * 300")
Out[38]: 3600

In [39]: eval("12 > 5")
Out[39]: True

In [40]: num = 4

In [41]: eval("num * 3")
Out[41]: 12

In [42]: eval("Hello, world")
Traceback (most recent call last):

  File "<ipython-input-42-df84d940b726>", line 1, in <module>
    eval("Hello, world")

  File "<string>", line 1, in <module>

NameError: name 'Hello' is not defined
```

Boolean expressions

"Boolean expressions" are expressions where the type of the result is a Boolean (True or False).

Relational operators allow defining Boolean expressions.

Relational operators

`<, <=, >, >=, ==, !=`

Applicable to operands of any kind.

If the comparison is true, the result is `True`.

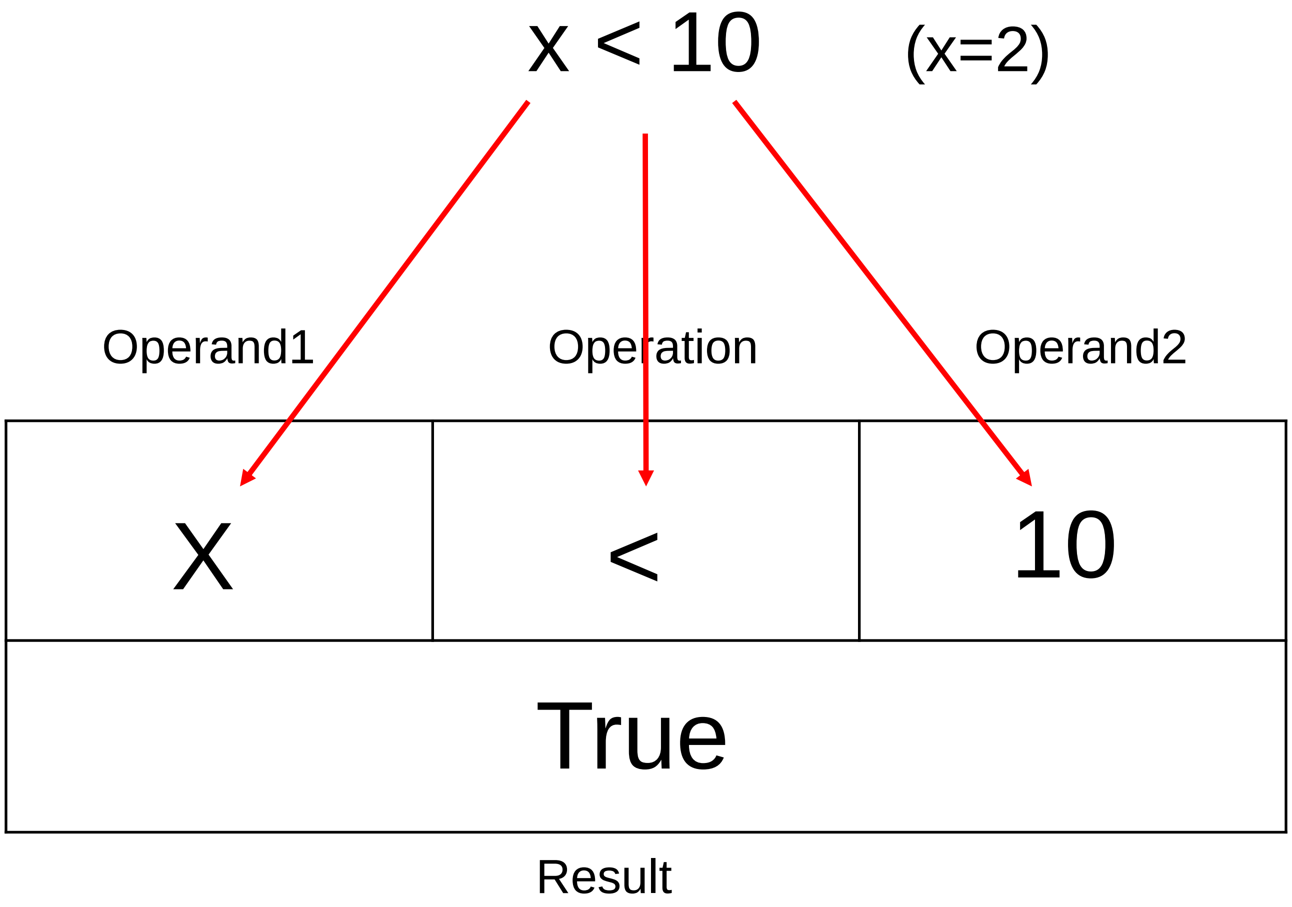
If the comparison is false, the result is `False`.

Examples:

Let's assume x is 3 (`x=3`):

`x < 5` → `True`	`x >= 3` → `True`
`x <= 2` → `False`	`x == 3` → `True`
`x > 3` → `False`	`x != 3` → `False`

Relational operators



x < 10 (x=2)

Operand1

Operation

Operand2

X

<

10

True

Result

In Python this is Ok

$0 < x < 10$

Although in many languages it is not possible.

Logical operators

They allow defining complex statements from relational operators.

Logical operators

Operation	Symbol	Example	Result
AND	and	(5 < 3) and (10 < 20)	False
OR	or	(5 < 3) or (10 < 20)	True
NOT	not	not(5 < 3)	True

and	True	False
True	True	False
False	False	False

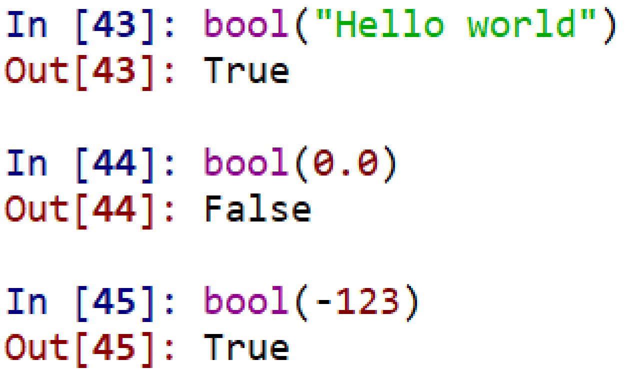
or	True	False
True	True	True
False	True	False

not	
True	False
False	True

Type Conversions

- Just as we could do type conversions between `str`, `int` and `float` types, we can also do conversions with the `bool` type.

- The `bool` function converts a value or variable to `False` or `True`. If the value or variable is equal to 0, it returns `False`, otherwise returns `True`.



```
In [43]: bool("Hello world")
Out[43]: True

In [44]: bool(0.0)
Out[44]: False

In [45]: bool(-123)
Out[45]: True
```

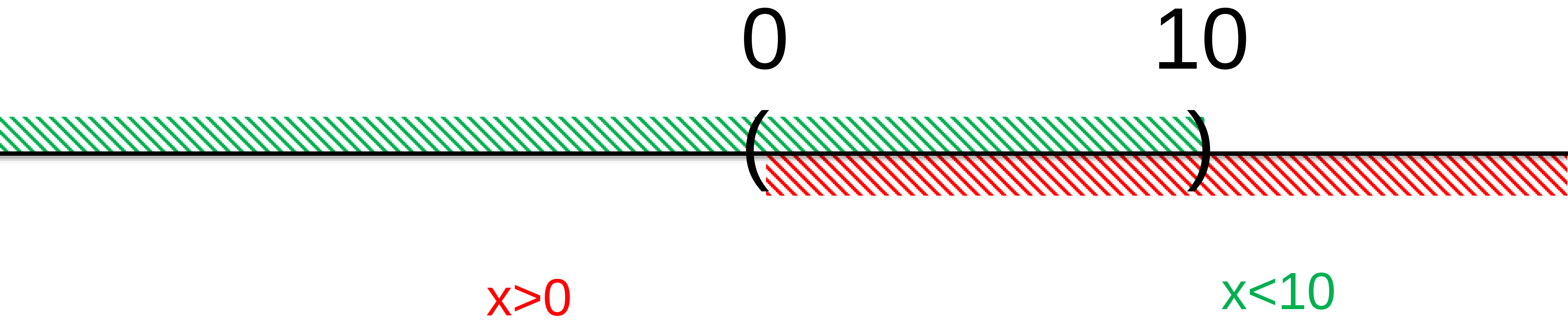
Exercise

Write the following program:

1- Ask the user for an integer value.

2- Display True if the value is between 0 and 10. Otherwise, display False.

$$0 < x < 10$$



Solution

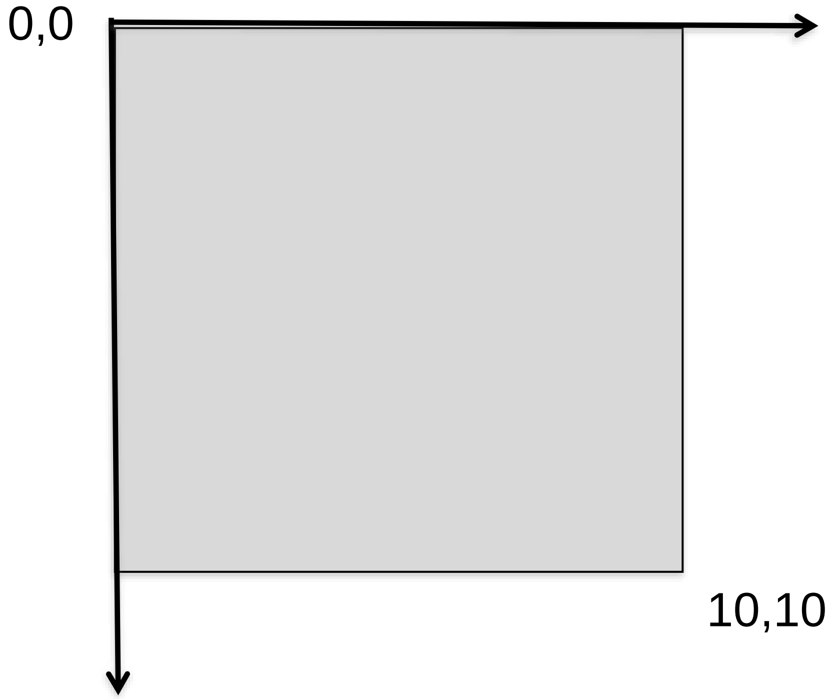
Remember that input() always reads as string (`str`)

```
x = int(input())
print((x > 0) and (x < 10))
```

Exercise

Write a program that ask the user to introduce two integers that are the x and y coordinates of a point in a 2D plane.

The program must print True if the point is inside a square of width and height 10. In case the point is out of the square, the program must print False.



Solution 1

```
x = int(input("Coordinate x: "))
y = int(input("Coordinate y: "))
print((x >= 0 and x <= 10) and (y >=0 and y<=10))
```

Solution 2

```
XMIN = 0
YMIN = 0
XMAX = 10
YMAX = 10

x = int(input("Coordinate x: "))
y = int(input("Coordinate y: "))
print((x >= XMIN and x <= XMAX) and (y >=YMIN and y<=YMAX))
```

Solution 3

```
XMIN = 0
YMIN = 0
XMAX = 10
YMAX = 10

x = int(input("Coordinate x: "))
y = int(input("Coordinate y: "))
print((XMIN <= x <= XMAX) and (YMIN <= y <= YMAX))
```

Membership operators

They are operators to assess whether one object is inside another one.

Membership operators

Operation	Symbol	Example	Result
Included	`in`	`“a” in “Hello”`	`False`
Not included	`not in`	`“la” not in “Hello”`	`True`