



15-110 PRINCIPLES OF COMPUTING – S19

LECTURE 4: VARIABLES

TEACHER:
GIANNI A. DI CARO

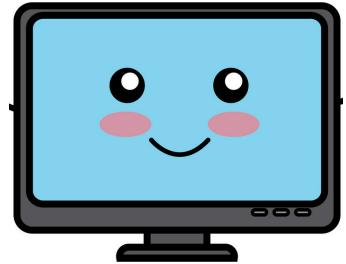
We need more than literals

- Just using literals (VALUES!) and print() function doesn't give us much freedom of doing things

- `2+3, 5.5 / 1.5`
- `print("Temperature is", 35.3, "degrees")`
- `print("This statement is:", False)`
- `((2+3) * (10 // 5) + (5 % 2)) / 2`



+

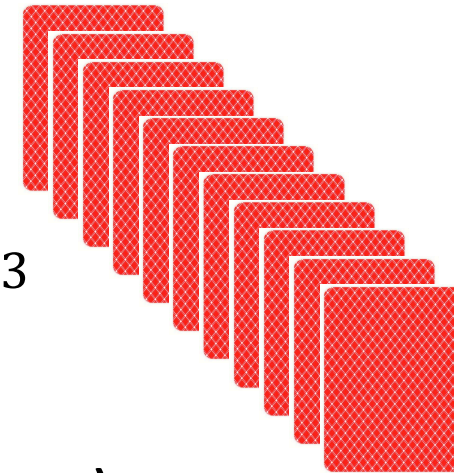


- We need to **store values, retrieve values, change values through operators, repeat operations** ... in order to design and implement (in Python) effective algorithms!

An algorithm for playing the one-pile NIM-11 game

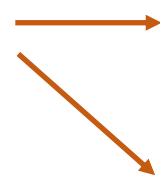
Rules:

- 11 items (e.g., matches, cards) are on the table
- Two players taking turn
- At each turn a player removes k items from the table, $1 \leq k \leq 3$
- The player taking the last item(s) loses the game



Algorithm (winning strategy for first player):

Storing, accessing,
updating values

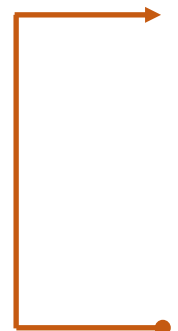


1. $state = 11$

2. Player 1: Remove two items

3. $state = 11 - 2$

iteration



4. Player 2: Remove k items

5. $state = state - k$

6. Player 1: Remove $4 - k$ items

7. $state = state - (4 - k)$

conditional

8. **Repeat** from 4 **until** there are no more items to remove

An algorithm for the calculation of the square root

Problem: Given a real number x , what is the value of \sqrt{x} ?

$$x = 4 \rightarrow \sqrt{4} = 2,$$

$$x = 2 \rightarrow \sqrt{2} = 1.414 \dots$$

- *Declarative knowledge:* Square root y of a number x is such that $y \cdot y = x$ (from $y = \sqrt{x}$, squaring both sides)

1. Start with an arbitrary guess value g : $y = g$

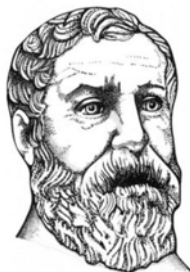
2. **If** $y \cdot y$ is close enough to x (with a given numeric approximation)

Then Stop, and say that y is the answer

3. **Otherwise** create a new guess value by averaging current guess y and x/y :

$$y = \frac{(y + x/y)}{2}$$

4. **Repeat** from 4 **until** $y \cdot y$ is close enough to x , and return y as the answer



Heron of Alexandria, 60 A.D.
(Babylonians, much earlier)

An algorithm for the calculation of the square root

■ $x = 25, \quad y = \sqrt{25} ?$

1. Start with an arbitrary guess value, $g = 3 \rightarrow y = g$

2. $(3 \cdot 3 = 9) - 25 \approx 0?$ No

3. New guess: $y = \frac{(3+25/3)}{2} = 5.66$

4. $(5.66 \cdot 5.66 = 32.04) - 25 \approx 0?$ No

5. New guess: $y = \frac{(5.66+25/5.66)}{2} = 5.04$

6. $(5.04 \cdot 5.04 = 25.4) - 25 \approx 0?$ Yes

$\rightarrow y = 5.04$

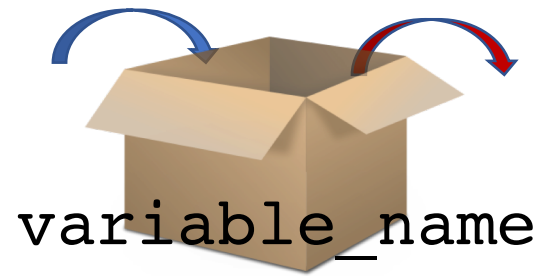
Guess-and-check algorithm

Variables

- In math we commonly use **named** parameters and variables to refer to symbols that will take values that we don't know yet

$$y = ax + bx + c$$
$$x = -b \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

- **Variables:** provide a way to name *information* and access and modify the information. by using the name



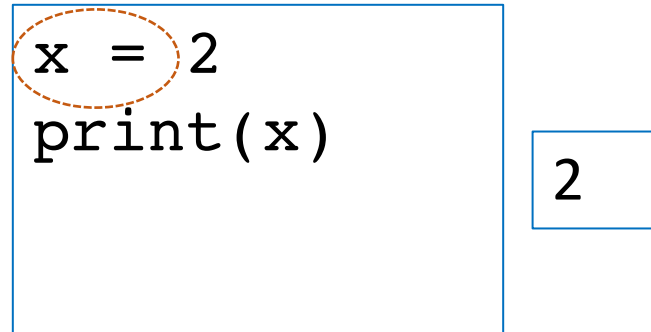
- A named *container* of information
 - What can we do with a variable (e.g., x, y)?
 - ✓ **Assign** its value $x = 2$
 - ✓ **Read / use** its value $y = x + 2$
 - ✓ **Modify** its value $x = 4.5$

- **Identifier:** a name given to an *entity* (a variable, a function, a class, ...)

Variables and assignment

- A literal is used to indicate a specific value, which can be *assigned* to a *variable*

- `x` is a variable and 2 is its value



The diagram consists of a large blue-outlined rectangle containing two lines of code: `x = 2` and `print(x)`. The first line, `x = 2`, is enclosed in a dashed orange oval. A dashed orange arrow points from this oval to the text '`x` is a variable and 2 is its value' in the list to the left. To the right of the large rectangle is a smaller blue-outlined rectangle containing the number '2'.

```
x = 2  
print(x)
```

2

Variables and assignment

- A literal is used to indicate a specific value, which can be assigned to a *variable*
- `=` is the **assignment operator**

- `x` is automatically defined as a variable and 2 is *now* its value

```
x = 2  
print(x)
```

2

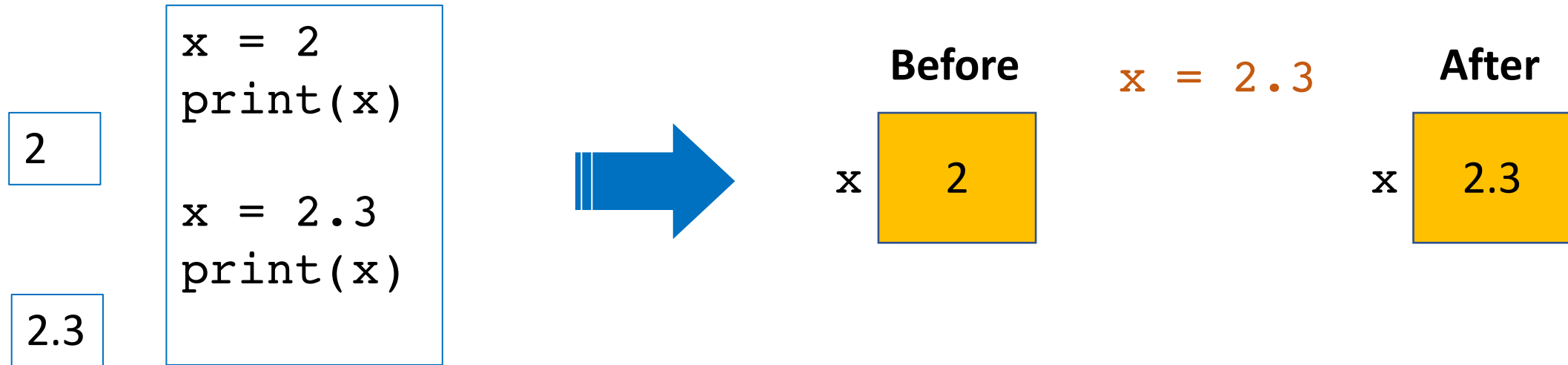
- `x` can be assigned different values; hence, it is a *variable*, its value can be “variable”

```
x = 2.3  
print(x)
```

2.3

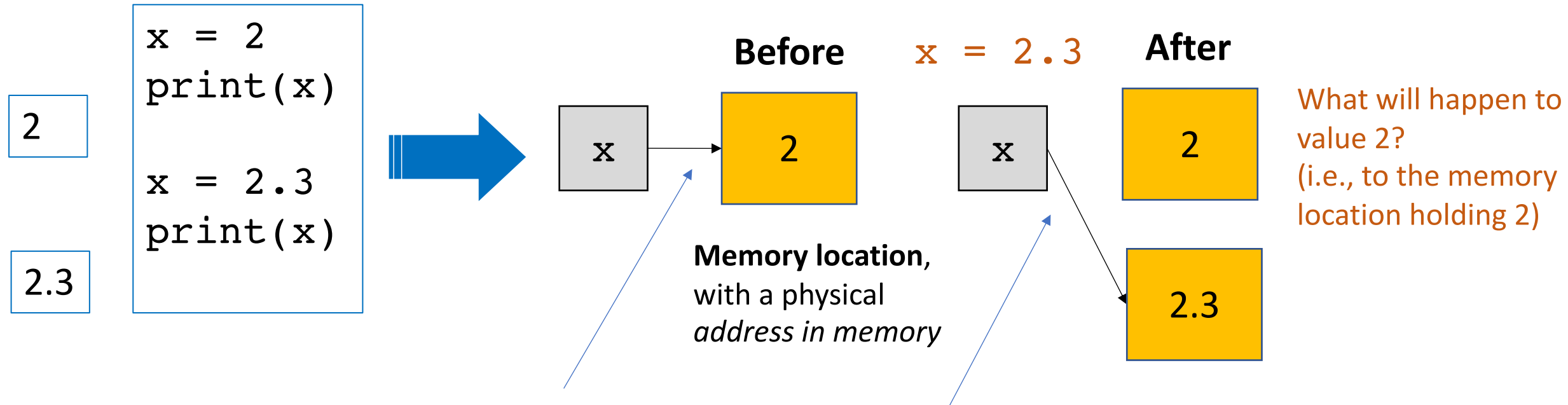
Box view of simple assignments

A simple way to view the effect of an assignment is to assume that when a variable changes, its old value is replaced in the named container



Real view of simple assignments

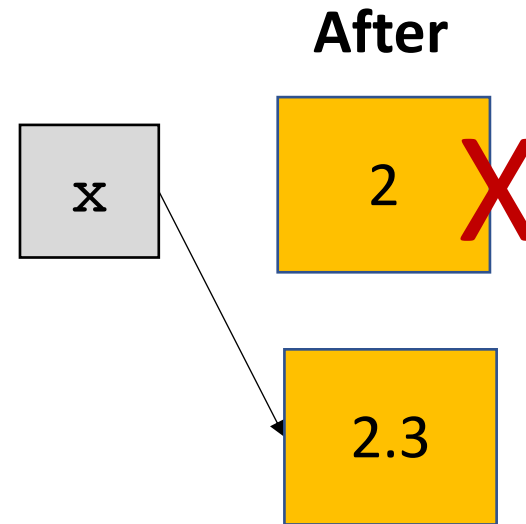
- The way the Python interpreter manages the assignment statements is actually more complex than the “variable as a (unique) box” model
- In Python, at the machine-level, **values *may* end up anywhere in memory**, and variables are just a high-level way used to refer to these memory locations



Variable `x` holds the **address** of the memory location where its value is *currently* held

Garbage collection

- ✓ Luckily, as a Python programmer you do not have to worry about computer memory getting filled up with old values when new values are assigned to variables
- ❖ Python will automatically clear old values out of memory in a process known as *garbage collection*



**Memory location
will be automatically
reclaimed by the
garbage collector**

Mutable vs. Immutable types

1. In the high-level python program:

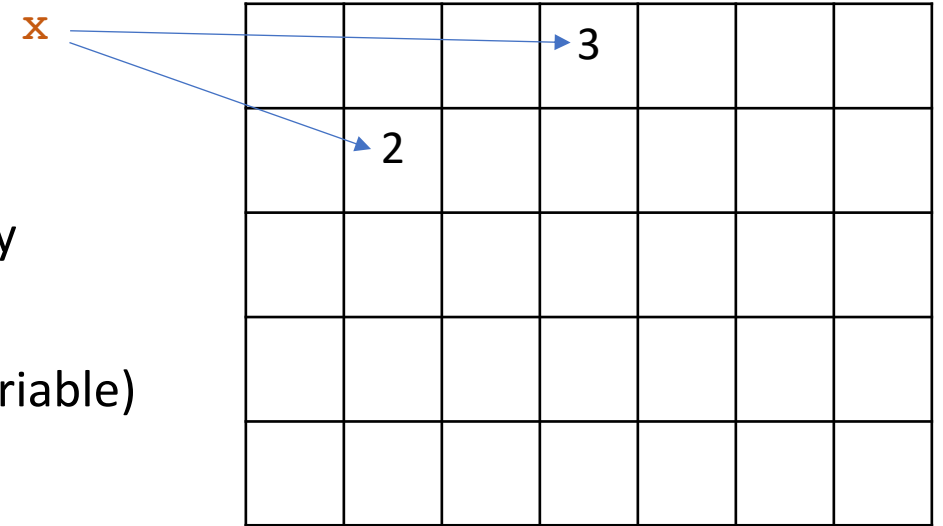
`x = 2` meaning that variable `x` has (`int`) value 2

- Internal to python / computer:

variable `x` holds the **reference** (the address) to the memory location to where its value 2 is currently *stored*

- The reference is associated to the identity `id()` of the variable)

`print(id(x))`



Memory locations

2. Next instruction in the high-level python program:

`x = 3` meaning that variable `x` has changed its (`int`) value to 3

- Internal to python / computer:

variable `x` is of immutable type `int`, meaning that *its value in memory cannot be changed*

→ A new memory location is allocated to hold the integer literal 3.

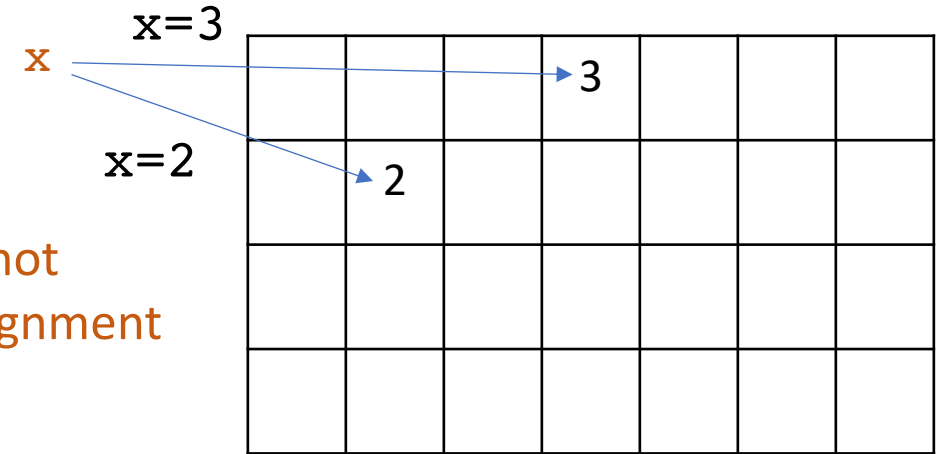
`x` now holds the reference to the new memory location (check `id(x)` !)

Mutable vs. Immutable types

Immutable data types:

- int
- float
- bool
- string
- function
- ...

The memory location holding the value will not (necessarily) be overwritten after a new assignment

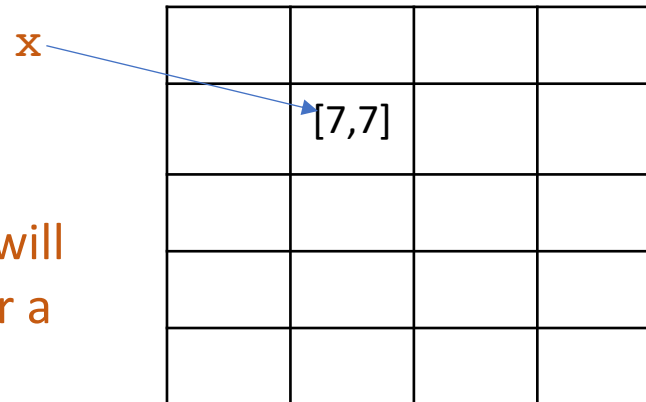


Mutable data types:

-

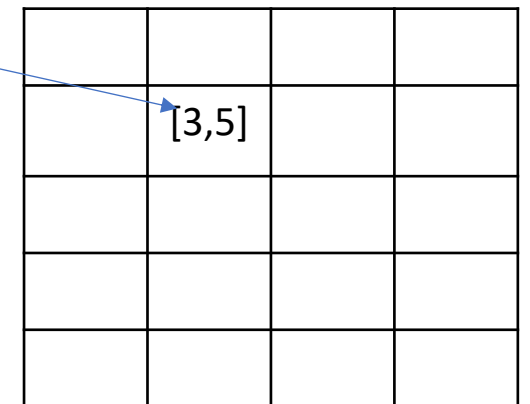
The memory location holding the value will be overwritten with the new values after a new assignment

$x = [7, 7]$



x

$x = [3, 5]$



Not a strongly-typed language!

- ✓ The type of a variable is set **at the moment of providing a value** to the variable (i.e., it is not necessary to *declare* the type of a variable before using it, as it happens in many other languages)
 - The command `x = 2` automatically sets the variable `x` as a variable of *integer type*
 - The command `x = 5.5` automatically sets the variable `x` as a variable of *float type*
 - The command `x = "Hello"` automatically sets the variable `x` as a variable of *string type*
- ✓ Type can change over the execution!
 - `x = 2`
 - `x = "I'm a string now"`
 - `x = True`
- Accordingly, also the memory location can (might *need to*) change in order to accommodate for the memory needs of the new type (e.g., non-scalar vs. scalar)
 - E.g., the memory space to hold an integer is not enough to hold a long string such as "I wouldn't fit in the four bytes used to hold an integer!"

Type Conversion and Type Casting

- We can also either **convert** or **cast** a type of an object into another!

```
x = 5      (type is int)
y = 2.5    (type is float)
x = y + 1   (now x's type is float)
print(x, y, type(x))
```

```
3.5 2.5 <class 'float'>
```

(Implicit) Type Conversion
(implicitly performed by
python interpreter)

(Explicit) Type Conversion: Casting
(explicitly performed by the user
with functions)

```
x = 5      (type is int)
y = 2.5    (type is float)
x = x + int(y)
print(x, y, int(y), type(x))
```

```
7 2.5 2 <class 'int'>
```

Type Casting built-in functions

- Type casting functions(with types known so far):

- `int(x)`: cast `x` to an **integer type**
- `float(x)`: cast `x` to an **float type**
- `bool(x)`: cast `x` to an **bool type**
- `str(x)`: cast `x` to an **string type**

Not all castings are feasible!

- A string type cannot be cast into a numeric one

- ❖ All these functions:

- **return the cast value**
- do not modify the value of the input variable `x`

```
x = 2.6
y = int(x)
print(x, y, type(x), type(y))
```


Allowed names for identifiers

Examples of allowed names for variables (and other identifiers)

```
house_color = "white"  
Color = 255  
House_color = 'yellow'  
HouseColor = "green"  
_HouseColor = 128  
X = 0.5  
x = 2  
Distance = 6 + _variable
```

```
distance = 5  
color = "red"  
x = 0.1  
night = False  
morning = True
```

```
house color = "white"  
house-color = "blue"  
0g = 2house2 + 1  
c.1 = 2  
C* = 0.5
```



NOT Allowed names

Rule of thumb: anything that could confuse the interpreter because of mixing up numbers, symbols and letters in the wrong place is not allowed

Reserved language keywords

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

Built-in functions (Python standard library)

<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

<https://docs.python.org/3/library/functions.html>

Style writing programs

- **Readability** and **clarity** of a program are essential, for you and for whoever will read / use your programs
- Moreover, **you'll be graded also based on readability and clarity of your homework!**
 - Even if you will pass all tests (i.e., your program is correct and does the expected job), you might lose points if the code is not *properly* written

```
a=2+3*4
b=2**a
c=float(b)
d="This is a mess"
e=10000//a+1
print(d,c,b,e)
```

Vs.

```
income = 2 + (3*4)
power_income = 2**a
real_power_income = float(b)
msg = "Income analysis"
fit_income_in_debt = 10000 // (a+1)
print(msg, '\n', "Income to power (real):", real_power_income, '\n',
      "Income to power (int):", power_income, '\n',
      "Integer fit of income into debt:", fit_income_in_debt)
```

First rules of thumb of writing clear and readable programs

- ✓ Give brief but self-explanatory names to variables and functions
 - E.g., a variable representing interest rate, shouldn't be called, a or b, but rather `interest_rate`
 - A variable representing a color, should be called `color`, not something (unclear) else
 - A function returning a DNA sequence, should be defined as `dna_sequence()`
- ✓ Use the form with lower letters and underscores to define variables' and functions' names (upper case letters will be used later on for defining classes, for instance, or global variables)
 - Use `car_velocity` instead of `CarVelocity` or `carVelocity`, or other possible forms
- ✓ Use spaces in statements
 - `x = 2 + y` is more readable than `x=2+y`, or, `print(x, y, z)` is better than `print(x,y,y)` ...
- ✓ Use comments using # and ''' to document your code if necessary (we'll see this next time)