15-110 Principles of Computing – S19

Lecture 2:
Python basics

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Road map

- Computer languages, compilers vs. interpreters
- Python!
- Objects, literals, and object types
- Basic operators for numeric types
- Variables and assignments
- Naming for identifiers
- Notion of function
- Display of results
- Type conversions and type casting
- Style notes

Programming languages

- A ***program*** is just a sequence of instructions telling the computer what to do: an encoding of an *algorithm for solving a problem*
- These instructions need to be expressed in a language that computers can understand
  - We refer to this kind of a language as a ***programming language***
  - Python, Java, C, C++ are notable examples of *imperative programming languages*
- A programming language is a type of *formal language* equipped with well-defined *syntactical rules*:
  - A set of well-defined **rules** to construct admissible words over a given alphabet of symbols (e.g., {a-z}, {A-Z}, {0,1,..9}, {+,-,*,/}, … {})
  - A set of well-defined **syntactic rules** (a *grammar*) to construct well-formed sentences (*expressions*) out of the admissible words
  - A **static semantics** that defines which syntactically valid expressions have meaning in the language
  - A **semantics**, which associates a meaning with each syntactically correct expression
- This is the same as a **natural language**, (e.g., English, Arabic) but more strict and precise!

(a formal language is defined by hand and reasoning, not *evolved by practice*!)

Programming languages (formal vs. natural)

- In English, symbols include letters $\{a - z\} \cup \{A - Z\}$, punctuation, accents, parentheses, …
- Rules (and practice) describe how to make admissible words
  - School ✓, ai:,?w ✗
  - *Python*: this_is_a_variable ✓, 0this_is:a_variable ✗
- Syntax rule describes which strings of words make (syntactically) well-formed sentences
  - "Cat dog student" it is not admissible since it is a form of *<noun> <noun> <noun>*
  - "You are CMU students" it is well-formed!
  - *Python:* 1.5 + 1.5 * a ✓, 1.5 1.5 a ✗
- Static semantics defines which syntactically valid strings of words have meaning in the language
  - "I have tired", "I are short", are both syntactically acceptable expressions since are in the form *<pronoun> <linking verb> <adjective>* but have not meaning in English
  - *Python:* 4.5 / "abc" is syntactically well formed (<literal> <operator> <literal>) but produces a static semantic *error* since it doesn't make sense to divide a number by a string of text!

Programming languages: semantics

- Semantics associate a meaning with an expression admitted by static semantics
  - In English, semantics can be **ambiguous**:
  - Look at the dog with one eye.
    - Look at the dog using only one of your eyes.
    - Look at the dog that only has one eye.
    - Perhaps the dog has found an eye somewhere, and we're looking at the dog.
  - In Python (and any other formal language) semantics of an expression is **uniquely defined**
  - → Each "legal" program has exactly one meaning! (one set of outputs, depending on the inputs
- Python's formal language is powerful enough to program (compute) anything that can be computed! (Python is **Turing complete**)
- Warning: YOU will have to take care of writing programs that are syntactically correct, complies with the static semantics of the language, and are semantically correct (they do what you expect to do!)

Machine Languages

- Python, Java, C, and C++ are, indeed, examples of *high-level* languages

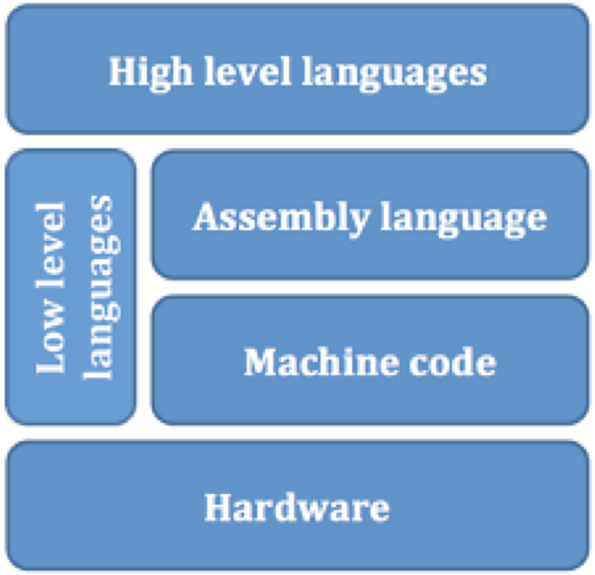
- Strictly speaking, computer hardware can only understand a very *low-level* language known as *machine language*

- If you want a computer to add two numbers, a and b, the instructions that the CPU will carry out might be something like this:

Load the number a from memory location 2001 into the CPU
Load the number b from memory location 2002 into the CPU
Add the two numbers in the CPU (using the ALU)
Store the result into the memory location 2003

A Lot of Work!

A hierarchy of languages



Levels of Programming Languages

High-level program

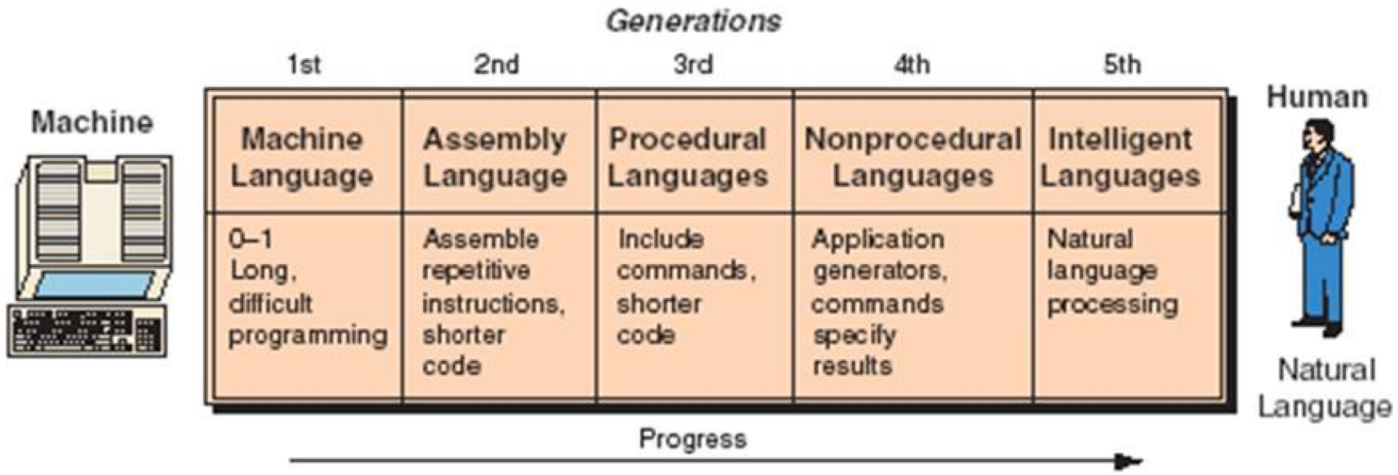
```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

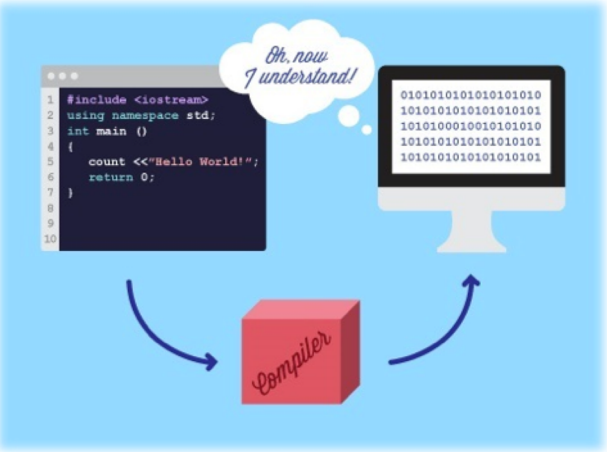
Executable Machine code

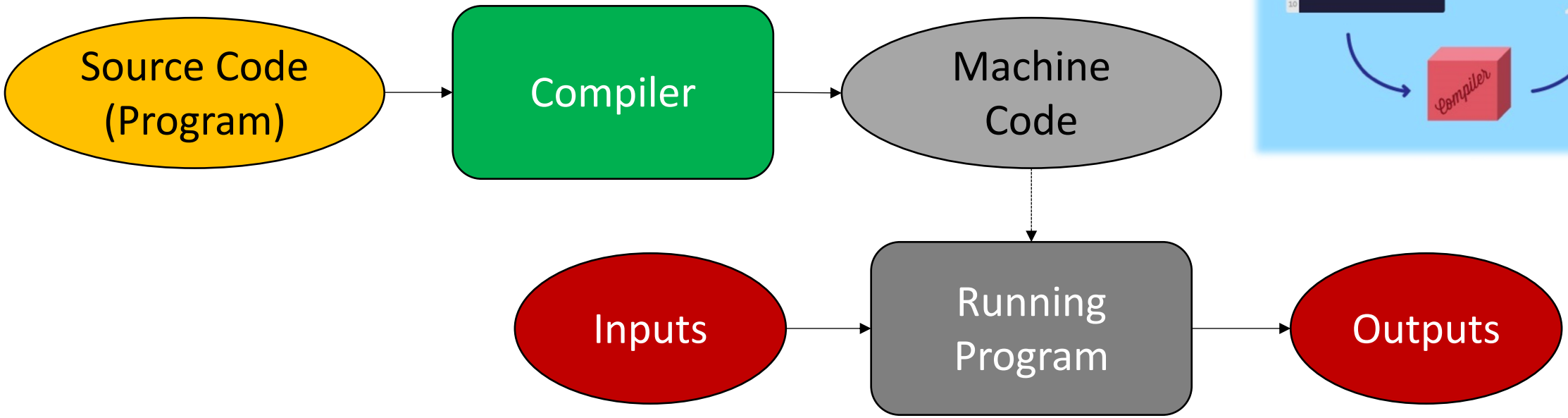
```
0001001001000101
0010010011101100
10101101001...
```



Compiling a high-level language

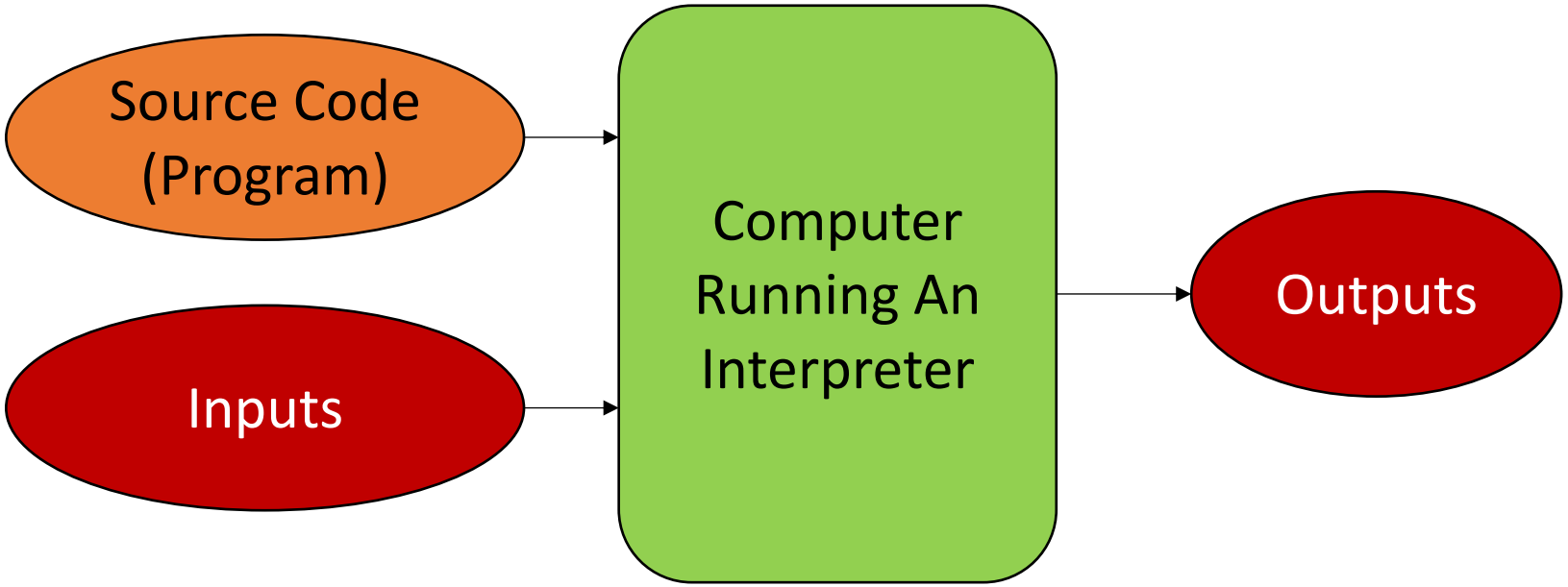
- A *compiler* is a complex software that takes a program written in a high-level language and *translates* it into an equivalent program in the machine language of some computer





Interpreting a High-Level Language

- An *interpreter* is a software that analyzes and executes the source code instruction-by-instruction (*on-the-fly*) as necessary



Python is an interpreted language

Compiling vs. Interpreting

- Compiling is a **static** (i.e., pre-execution), one-shot translation
  - Once a program is compiled, it may be run over and over again without further need for the compiler or the source code
- Interpreting is **dynamic** (i.e., happens during execution)
  - The interpreter and the source code are needed every time the program runs
- Compiled programs tend to be faster, while interpreted ones lend themselves to a more flexible programming environments (*they can be developed and run interactively*)

Note on Portability

- The translation process highlights another advantage that high-level languages have over machine languages, namely, *portability*
- A program for an Intel-based machine will not run on an IBM-based machine since each computer type has its own machine language
- On the other hand, a program written in a high-level language (say, a Python program) can be run on many different kinds of computers as long as there is a suitable compiler or interpreter
- Python programs are said to be portable!

Let's start with Python!


```
3.5 + 2
x = 3
y = 2 * 2.1
print(x + y + 1)
msg = "this is a simple program"
print(msg)
```

- A python program (also termed a *script*) is a sequence of definitions and commands
  - Definitions are *evaluated*
  - Commands (also termed statements) are *executed* (one at-a-time) by the python interpreter
    - Commands instruct the interpreter to do something
    - A command is the smallest standalone element of an imperative program
    - Statements can include expressions, that combine objects and operators
    - Command execution happens within a *shell*, an interface to the OS
    - When a new program execution begins, a new shell is being created

Objects

- **Objects** are the basic entities that Python manipulates (building blocks of information handling)
- Every object has a type that defines the things that can be done (or not) with the object
- *Type?* We said that a computer is a (phenomenal) number cruncher, therefore, the basic building blocks for performing computation must be numbers …
  - + textual strings to conveniently represent textual data
  - + logical (binary) data to conveniently represent truth of falsity of conditions (for flow control!)
  - + additional *structured ways* to frame and represent numbers, strings, and logical data …

- Objects can have a *name* (identifier) or not (be just *values*), and in this case are termed literals
- Types can be either scalar (*indivisible*), or not scalar (*composite*)

Scalar types

- Scalar type literal objects:
  - `int`: **Integer relative numbers ($\mathbb{Z}$)**
    - Examples of literals of type `int` are: 2, 3, -1, 1000, 2001, -99
  - `float`: **Real numbers ($\mathbb{R}$)**
    - Examples of literals of type `float` are: 2.0, 3.2, -1.5, 1000.0, 2001.002, -99.1, 1.6E3
    - Why are do they called *float* instead of *real*?
  - `complex`: **Complex numbers ($\mathbb{C}$)**
    - Examples of literals of type `float` are: 2.0+3j, 3+j, -1.5-5j
  - `bool`: **Boolean (logical) values**
    - Instances of literals of type `bool` are: `True, False`
  - `None`: **Type with a single value**
    - Instance of a literal of type `None` is: `None`

Non-Scalar types

- Non-Scalar type literal objects: (we will see much more of these next week and later on!)
  - `str`: **String of characters (non-numeric text)**
    - Examples of literals of type `str` are:

      “Hi”, “abc”, “Hello!”, ‘z’, ‘abc’, ‘_wow_’, “I’m Joe”, ‘ Say “hello!” to her’
  - `tuple`
  - `list`
  - `set`
  - `dict`

Operators

- **Operators** can be used to perform operations on objects based on their data type
  - Objects → *Operands*
  - *Operator* → Action to be executed on the operands
  - Two literals: 2 and 3, operator + → 2 + 3 Infix notation (typical in arithmetic)
  - We could write it in other ways:
    - + 2 3 Prefix notation (Polish notation)
    - 2 3 + Postfix notation (Reverse Polish notation)
- Objects and operators, when combined form **expressions**
- In turn, each expression denotes an object of some type, which is the **value** of the expression
  - 5 is the value of 2 + 3, and it has type `int`
  - What is the value and type of 2.2 + 3?

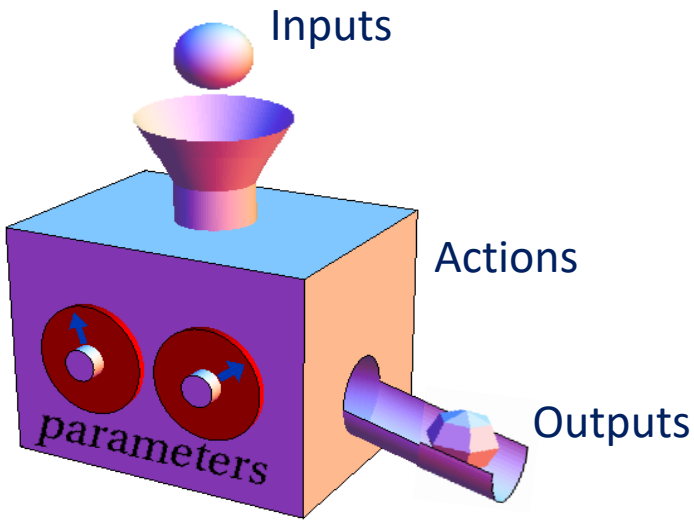
Operators for numeric types

- Let `i` and `j` be two literals that can be either `int` or `float`
- **Sum:** `i+j,` type of expression: `int` if both integers, `float` otherwise
- **Difference:** `i-j,` type of expression: `int` if both integers, `float` otherwise
- **Product:** `i*j,` type of expression: `int` if both integers, `float` otherwise
- **Integer division:** `i//j,` returns the integer quotient and ignores the real remainder, type of expression: `int` if both integers, `float` otherwise
  - $i \div j = j \cdot n + r$ where n is an integer number (returned), and r is a real number
- **Division:** `i/j,` returns the real-valued result of the division, type of expression: `float`
- **Modulus:** `i%j,` returns the remainder from the division of the first argument by the second, real-valued result of the division, type of expression: `float`
  - $i \div j = j \cdot n + r$ where n is an integer number, and r is a real number, which is returned
  - What is the result of `i%j` when `i<j`? (e.g., 2 % 6)
- **Power raising:** `i**j,` returns the i^j , type of expression: `int` if both integers, `float` otherwise

Functions (something preliminary on functions …)

- What could we do with what we have so far?
  - Perform some arithmetic operations (including the use of parentheses of precedence rules)
    - 2**3
    - 4.5 // 3
    - (1+2)*3
  - **Display** (know) the result (the value) of an expression!
    - print(2**3)
    - print(4.5//3)
  - **Know the type** of an expression (and of the final result)
    - type(2**3)
    - type(4.5//3)
  - What those `print()` and `type()` are? → **Functions!**

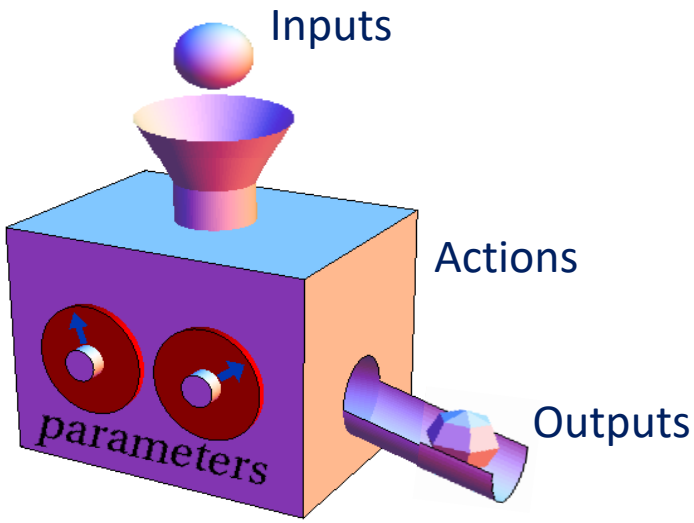
Functions



- **Functions** provide a way to refer to a procedure, a series of actions, to be executed whenever the function is called in the program
- A function performs some useful service
- Python has a number of built-in functions: ready to use
- Functions can be imported from external modules
- *Custom functions* can be newly programmed to pack in the code of the function a set of relatively complex actions that perform a desired service

- Functions can **require or not input parameters (arguments)**:
  - cubic_root(9), move_robot_for_a_distance(10), Pythagoras(3, 4), print(4), wake_me_up()
- Functions can **return or not a value (of a specified type):**
  - Pythagoras(3, 4) would return the hypotenuse value as a float
  - Is_everything_ok() returns a Boolean
  - Move_robot() doesn't return any value, it only performs the action, the same as print(4)

Functions



- **Custom Functions** can be defined as follows

```
def give_me_five():
    return 5

def make_sum (x,y):
    return x+y

def do_nothing ():
    return
```

Python language (reserved) **keywords**

- **def**
- **return**

Indentation and colons (:) matter!
Use TAB for indenting!

Let's go back to use print() and type() …

We need more than literals ….

- Just using literals doesn't give us much freedom of doing things
- We need to **store values, retrieve values, change values**, … *over and over*, like in the seen algorithm for finding the square root of a number
  - Need to store and retrieve the value of x and the values assigned to g, starting from the first assignment (that would set it to a float)

- *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 **If** $g \cdot g$ is close enough to x (with a given numeric approximation)

   Then `Stop`, and say that g is the `answer`

2. **Otherwise** create a new guess value by averaging g and x/g:

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

3. **Repeat** the steps 2 and 3 until $g \cdot g$ is close enough to x

Variables

- In math we use named parameters and variables, $a, b, c, \ldots x, y, z$ …

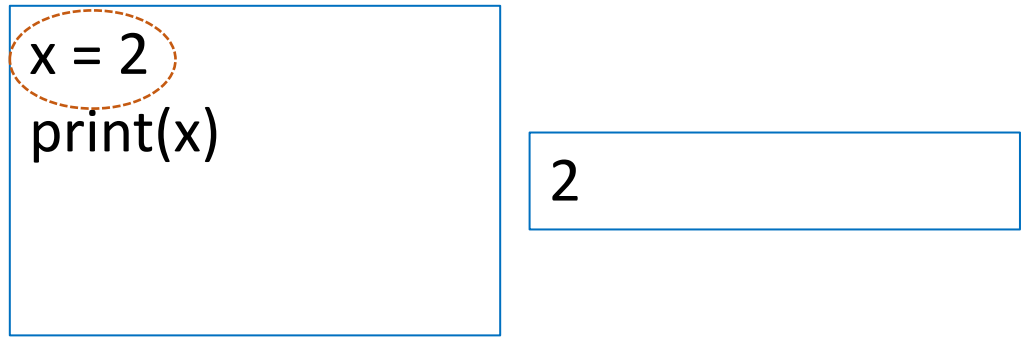
- **Variables:** provide a way to name, access, and modify *information*
- A named *container* of information

  - What can we do with a variable?
    - ✓ **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, …)

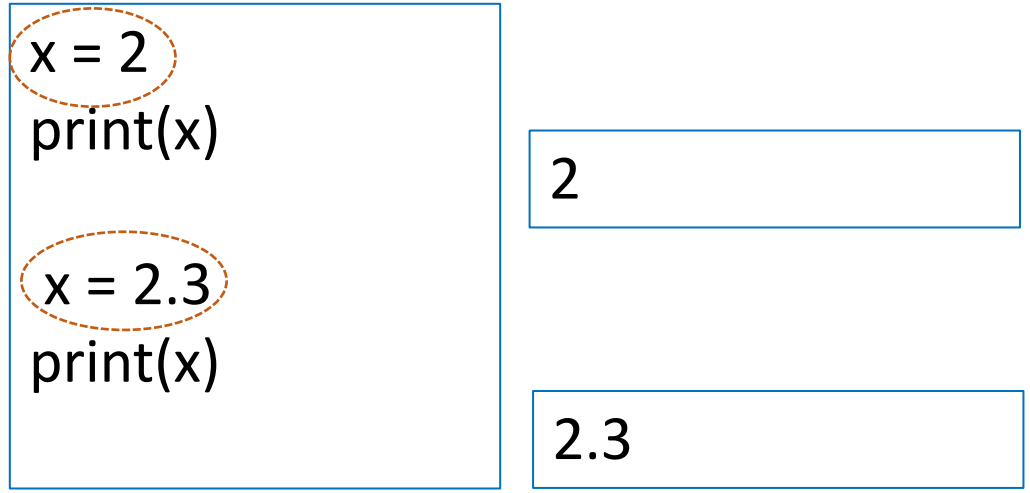
Variables and assignment

- 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



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
  - x can be assigned different values; hence, it is a *variable*



Box view of simple assignments

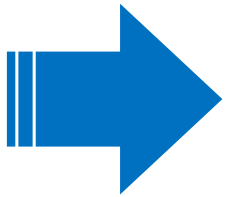
- A simple way to view the effect of an assignment is to assume that when a variable changes, its old value is replaced

2

2.3

```
x = 2
print(x)

x = 2.3
print(x)
```



x = 2.3

Before

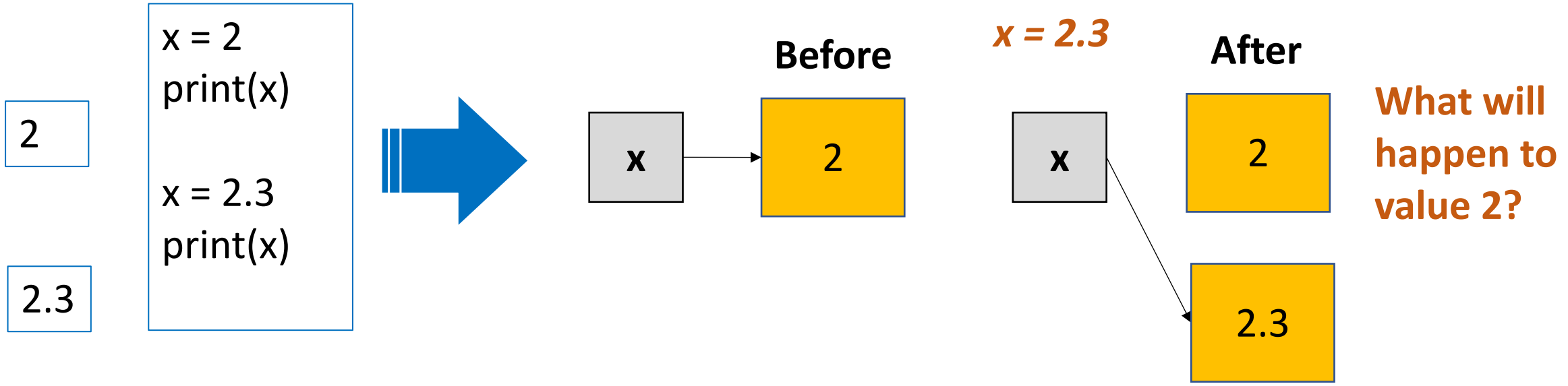
x 2

After

x 2.3

Real view of simple assignments

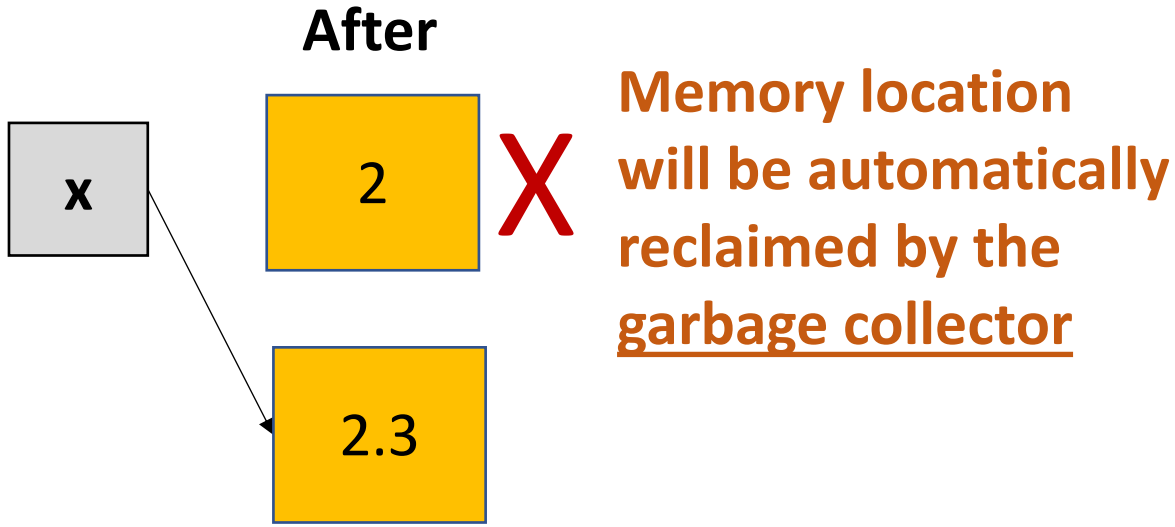
- Python assignment statements are actually slightly different from the "variable as a box" model
  - In Python, values may end up anywhere in memory, and variables are used to refer to them



Garbage collection

- Interestingly, 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*



Not a strongly-typed language!

- The type of a variable is set at the moment of providing a value to the variable
- In fact, it is not necessary to declare the type of a variable (and it's not immutable)
- Type can change over the execution
- And, accordingly, also the memory location can change in order to accommodate for the need of the new type (e.g., non-scalar vs. scalar)
- We can also **convert** or **cast** a type into another!

Type Conversion and Type Casting

```
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)

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

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

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

Allowed names for identifiers

Types

distance = 5 Integer number

color = “red” Text string

x = 0.1 Real number

night = False Boolean value (true / false)

morning = True

Allowed names

house_color = “white”

Color = 255

House_color = ‘yellow’

HouseColor = “green”

_HouseColor = 128

X = 0.5

x = 2

Distance = 6 + _variable

NOT Allowed names

house color = “white”

house-color = “blue”

0g = 2house2 + 1

c.1 = 2

C* = 0.5

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	

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
- *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)