15-110 Principles of Computing – S19

Lecture 3:
Python basics

Teacher:
Gianni A. Di Caro

Let’s start with Python 3!

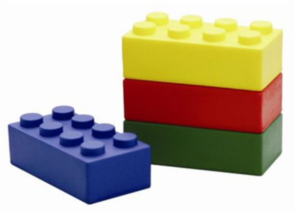
- A python program (also termed a *script*) is a <u>sequence</u> of definitions and commands
  - <u>Definitions</u> are ***evaluated***
  - <u>Commands</u> (also termed <u>statements</u>) are ***executed*** (one at-a-time) by the python *interpreter*
    - Command execution happens within a ***shell***, an interface to the OS
    - When a new program execution begins, a new shell is being created

- Commands *instruct* the interpreter to do something
- A command includes **Objects** (words and concepts)
- A command can also include **Operators** to act upon the objects and create **Expressions**

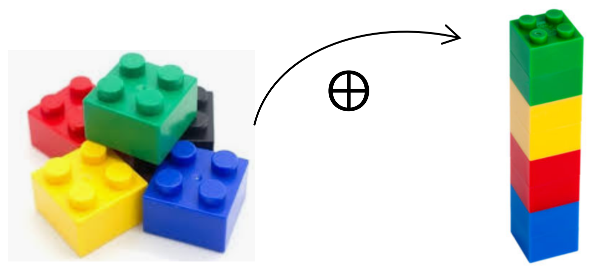
```
3.5 + 2
x = 3
y = 2 * 2.1
print(x + y + 1)
msg = "this is a simple program"
print(msg)
```

Terminology: Objects (literal, variable), Operators, Expressions

Objects: the basic entities that Python manipulates (building blocks of information handling)

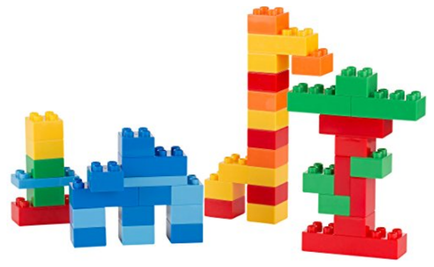


Operators: act upon the objects



Expressions: combine <u>objects *and* operators</u> together (to create new objects)

Command: an instruction to the interpreter, it features objects and, possibly, expressions



<u>Objects can be of two kinds:</u>

- ***Literal* objects** ↔ Only have a ***value*** (e.g, 1, 3.5)
- ***Variable* objects** ↔ Have a value and a ***name***, an ***identifier*** (e.g., x = 1, y = 3.5, table="red")

Terminology: Object types (numeric, string, logical, …)

Objects have a **type,** that defines the things that can be done (or not) with the object:
(e.g. with a car you can travel but not fly, with a cake you can eat but not travel ….)

- Numeric: to represent numbers of various type (e.g., `1, 3.5`)
- Character string: to represent textual information (e.g., `“Temperature is 35 degrees”`)
- Logical (binary) to represent truth of falsity of conditions (i.e., `True, False`)
- *Structured ways* to frame and represent groups of numbers, strings, and logical data …

Terminology: Scalar vs. Non-Scalar objects

An object type can be **composite**, made of multiple components, or be **indivisible**

- Scalar type (e.g., `1, 3.5`)
- Non-scalar type (e.g., `“Hello”`)

Vs.

✓ **Scalar**: the object is indivisible

✓ **Non-scalar**: the object is composed by multiple parts that can be individually manipulated (accessed, modified, removed, added)

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*?
  - `bool`: **Boolean (logical) values**
    - Instances of literals of type `bool` are: `True, False`
  - `complex`: **Complex numbers ($\mathbb{C}$)**
    - Examples of literals of type `float` are: 2.0+3j, 3+j, -1.5-5j
  - `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
  - For example, two literals: 2 and 3, operator + → 2 + 3 **Infix notation** (typical in arithmetic)
    - Note that 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
- **Division:** `i/j`
  - Returns the real-valued result of the division, type of expression: `float`
- **Power raising:** `i**j`
  - Returns the i^j , type of expression: `int` if both integers, `float` otherwise

Operators for numeric types

❖ Division between two numbers i, j: $i \div j = n + \frac{r}{j} \rightarrow i = j \times n + r$

where n is the integer quotient and r is the remainder of the integer division

$n =$ how many times j precisely fits in i,

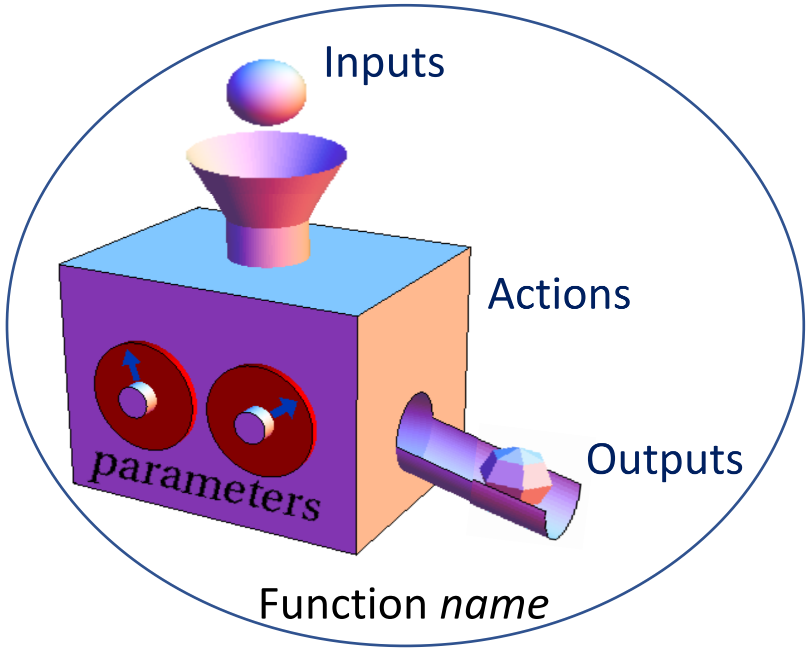
$\frac{r}{j} =$ the fractional remainder after the integer division

- **(real) Division:** `i/j` = $n + \frac{r}{j}$
- **Integer division:** `i//j` returns the integer quotient, n, and ignores the fractional remainder
  - Type of expression: `int` if both integers, `float` otherwise
- **Modulus:** `i%j` returns the remainder r from the *integer division*
  - Type of expression: `int` if both integers, `float` otherwise
  - What is the result of `i%j` when `i<j`? (e.g., 2 % 6)

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!**

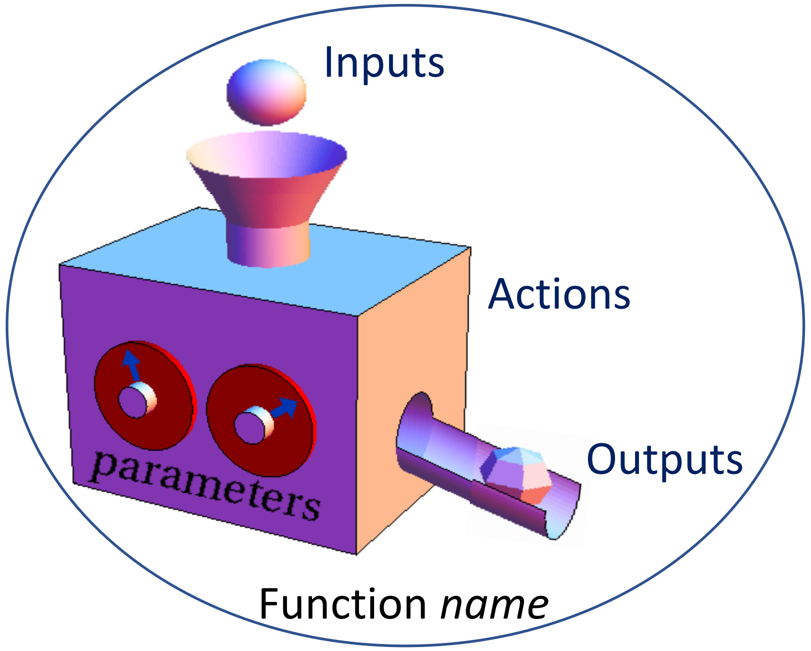
Functions



- **Functions** provide a way to refer <u>by name</u> (*identifier*) to a procedure, a series of actions, to be executed whenever the function is called in the program. A function is an object of type *function*
- A function performs some <u>useful service</u>
- Python has a number of <u>built-in functions</u>: ready to use

- Functions can be also <u>imported</u> from external modules
- <u>*Custom functions*</u> can be newly programmed to pack in the code of the function a set of relatively complex actions that perform a desired service

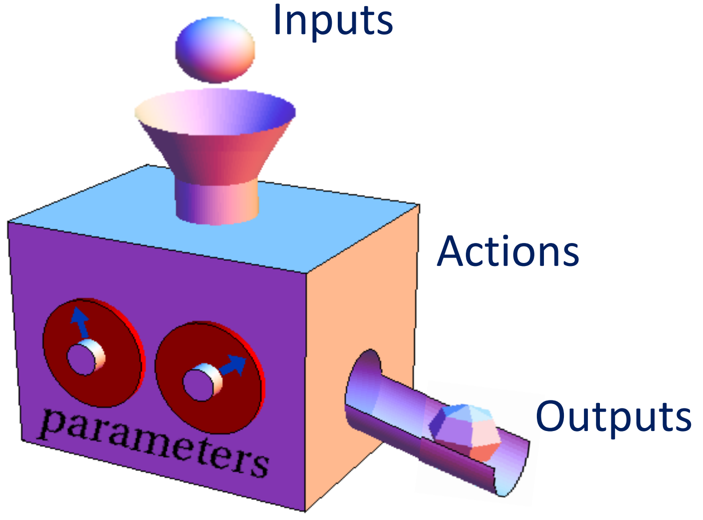
Functions



- Functions can **require or not input parameters (arguments)**:
  - `cubic_root(9)`
  - `move_robot_for_a_distance(10)`
  - `move_robot_for_predefined_distance()`
  - `Pythagoras(3, 4)`
  - `print("Error!")`
  - `wake_me_up()`
  - `play_beep()`

- 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()` would return a Boolean
  - `move_robot()` would only performs the action, the same as `print(4)`

Functions



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

```
def give_me_five():
    return 5

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

def avg(x, y):
    z = 1/2
    return z * (x+y)
```

Python language (reserved) **keywords**

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

```
def error_msg (msg):
    print("Error:", msg)
```

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

> Let's go back to use print(), type(), operators, …
> → Check the Python notebook!