Python: variables, operators, basic input/output

Intro. to Computing

`https://mandarm.github.io/computing.html`

Pseudocode to Python

Example: problem 1

Pseudocode

I Initially, set variables X and Y to 0.

II **for** i in $\{1, 2, 3, \ldots, T\}$
$X \longleftarrow X +$ _____.

III **for** j in $\{1, 2, 3, \ldots, D\}$
$Y \longleftarrow Y +$ _____.

IV **If** X _____ Y
then print "*A partially empty reservoir never fills up if all taps and drains are kept open.*"
else print "*An initially empty reservoir fills up in* _____ *minutes if all taps and drains are kept open.*"

V **If** X _____ Y
then print "*A partially full reservoir never becomes empty if all taps and drains are kept open.*"
else print "*An initially full reservoir becomes empty in* _____ *minutes if all taps and drains are kept open.*"

Problem 1: Python equivalent

```
total_water_added_per_min = 0
total_water_removed_per_min = 0

for i in range(num_taps):
    total_water_added_per_min += 1/filling_time

for i in range(num_drains):
    total_water_removed_per_min += 1/draining_time

if total_water_added_per_min == total_water_removed_per_min:
    print("A partially full reservoir never becomes completely full or
     empty when all taps and drains are kept open.")
elif total_water_added_per_min > total_water_removed_per_min:
    print("An initially empty reservoir becomes completely full in",
          1 / (total_water_added_per_min - total_water_removed_per_min),
          "minutes.")
else:
    print("An initially full reservoir becomes completely empty in",
          1 / (total_water_removed_per_min - total_water_added_per_min),
          "minutes.")
```

More meaningful names than X, Y

`range(N)` $\equiv 0, 1, \ldots, N-1$ (more later)

X `+=` $Y \equiv X \leftarrow X + Y$

Later: try replacing comma (`,`) with `+`

Algorithms/pseudocode to programs

What did we need?

- **Variables:** to store values, and to use them in calculations
- **Operators** (and functions): to do addition, etc., and other actions like printing output
- **Statements:** putting variables, operators and functions together to executing actions
  - assignment statements (=, +=)
  - conditional statements (`if`)
  - loops (`for`)

What are we still missing?

- **Input methods:** how do we get the values provided by the user?

Problem 1: missing pieces

Command line arguments

```
import sys

if len(sys.argv) != 3:
    sys.exit('Usage: ' + sys.argv[0] + ' <number of taps> <number of
    drains>')

num_taps = int(sys.argv[1])
num_drains = int(sys.argv[2])
```

Using `input()`

```
    message = 'How long does it take for Tap ' + str(i) + \
        ' to fill an empty reservoir by itself? '
    filling_time = int(input(message))
```

Running Python

Interactively via the (I)Python interpreter

- Run `$ python3` **or** `$ ipython`

```
$ python
Python 3.14.6 (main, Jun 15 2026, 11:36:54) [GCC 16.1.1 20260430] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
$ ipython
Python 3.14.6 (main, Jun 15 2026, 11:36:54) [GCC 16.1.1 20260430]
Type 'copyright', 'credits' or 'license' for more information
IPython 9.16.0 - An enhanced Interactive Python.  Type '?'  for help.
Tip:  Use `%timeit` or `%%timeit`, and the `-r`, `-n`, and `-o` options to
easily profile your code.

In [1]:
```

- Type Control-D (Linux/WSL) or Control-Z (Windows?) or `quit` or `exit` to exit.

Running “stored” programs

- Write your program in a file and run it

```
$ python <your-file-name> <command-line arguments if any>
```

Exercises

Download `exercises1.py` from the course page, and execute the commands / statements / lines in the file **one by one**, at the Python interactive prompt (running the entire file at once will result in early termination because of errors). Observe and understand the error messages printed or output generated, as well as the differences between different ways of doing almost the same task.

Python basics

Variables

- Name assigned to memory location or the value stored in it
- Assignment is said to *bind* a name to a value
- Each variable has a *type*
- Variable's type defines operations permitted on that variable
- Same variable (name) can be bound to different values, possibly of different types
  $\Rightarrow$ variables are said to have *dynamic* types

Exercises:

```
x = 10
type(x)
x = "Hello"
type(x)
```

`type()` function returns the type of a variable / expression

Variables: some terminology

- **binding:** current mapping (if any) for a name to a location
- **l-value:** any expression that corresponds to a location and can thus occur on LHS of assignment
- **r-value:** any expression that can occur on RHS of assignment

All l-values are r-values, but the converse is not true.

```
a = 10       # binds a to the value 10
b = a + 5    # binds b to the value 15
```

- `a` $\rightarrow$ l-value in line 1, r-value in line 2
- `10`, `5` $\rightarrow$ r-values, **not** l-values

Variables: syntax

- Names must start with a letter (uppercase/lowercase, i.e., `A-Z`, `a-z`) or *underscore* `_`.
- Names may contain letters, underscores and digits (`0-9`, anywhere other than in the initial position)
- No upper length limit

Keywords / reserved words: cannot be used as ordinary identifiers

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

Also `match`, `case`

Reserved classes of identifiers (LATER)

Specific patterns of **leading** and **trailing** underscore characters
$\Rightarrow$ identifiers with special meanings

- Names in module starting with underscore: not imported by `from module import *`
- **In interactive interpreter**, single underscore $\equiv$ result of the last evaluation
- **In programs**, no special meaning, but *commonly used* for unused variables
- Names starting **and** ending with __ ("dunder" names): system-defined names; do not use in your programs
- Names starting with __: class-private names

Basic types

- `int` `783` `0` `-192`
- `float` `1.25` `1.2e10` ($\equiv 1.2 \times 10^{10}$) `1.2E-2` ($\equiv 1.2 \times 10^{-2}$)
- `bool:` only two permitted values `True` `False`
  - `0` and `None` $\equiv$ `False`
- `str` sequence of *characters* (Unicode allowed)
  - delimiters: `"..."` also `'...'` `"""..."""`
  - multiline strings allowed with `"""`, e.g.

```
"""String
across
lines"""
```

  - **almost all input is read as a string**
- `NoneType` only permitted value `None`

Later: binary, octal and hexadecimal integers

Type conversion or casting

`int()` `float()` `bool()` `str()`

Exercises:

1. Try the following commands at the Python prompt (in order). As before, observe and understand what is happening.

```
s = "1.9e0"        bool(None)      "Hello" + 2
float(s)           bool(-100)      "Hello" * 2
int(s)             bool(144.55)    2 * "Hello"
int(float(s))      bool(0)         "2" * "Hello"
str(1.9e-10)       bool("hi")      "2" + "Hello"
str(1.9e-1)
```

2. Try to work out experimentally when `str()` prints floating point numbers in regular form, e.g., `0.01`, and when it uses the 'scientific' notation, i.e., `1.2e-1`.

Basic operators

- **Arithmetic operators**: `+` `-` `*` `/` `//` `%` `**`
  - `/` is division, e.g., `1/2` → `0.5`
  - `//` is *integer division* or quotient, e.g., `1//2` → `0`
  - `**` is exponentiation, e.g., `2**4` → `16`
- **Comparators** or **relational operators**: `<` `>` `<=` `>=` `==` `!=`

  works for `int`, `float`, `string` types

  Later: the `is` operator
- **Logical** or **Boolean operators**: `and` `or` `not`
  - condition1 `and` condition2 is `True` if and only if **both** condition1 and condition2 hold
  - condition1 `or` condition2 is `True` if and only if **at least one** of condition1 and condition2 holds
  - `not` condition

Operators: precedence

Precedence and associativity: rules for evaluating expressions when brackets are not given

Examples:

```
10 - 4 - 3
10 - (4 - 3)
2 ** 2 ** 3
4 + 2 ** 3
2 > 3 or 1 < 2
2 < 3 and 4 > 5 or 1 < 2
```

Precedence table:

(incomplete, descending order)

`**`	exponentiation
`+x  -x`	positive, negative
`*  /  //  %`	
`+  -`	
`<  <=  >  >=  !=  ==`	
`not x`	Logical/Boolean NOT
`and`	Logical/Boolean AND
`or`	Logical/Boolean OR

Associativity: group left to right, **except for exponentiation**

Exercises

Remember what you observe!

1. Try the quotient (`//`) and remainder (`%`) operators with negative numbers.
2. *Lexicographic ordering:* try the comparison operators on strings like `"a"`, `"aaa"`, `"ab"`.

Basic input / output

- Read using `s = input("Prompt string:  ")`
  - Prompt string is optional
  - `s` is always a string
  - Convert `s` to appropriate type using `int()`, `float()`, `bool()`, etc.
- Print using `print()`
  - `print()` prints just a newline
  - accepts multiple arguments of different types

Examples:

```
num = input("Type a number... ")
print(3*num)
print("Value of num is " + num)
num = int(num)
print(3*num)
print("Value of num is" + num)
print("Value of num is", num)
```