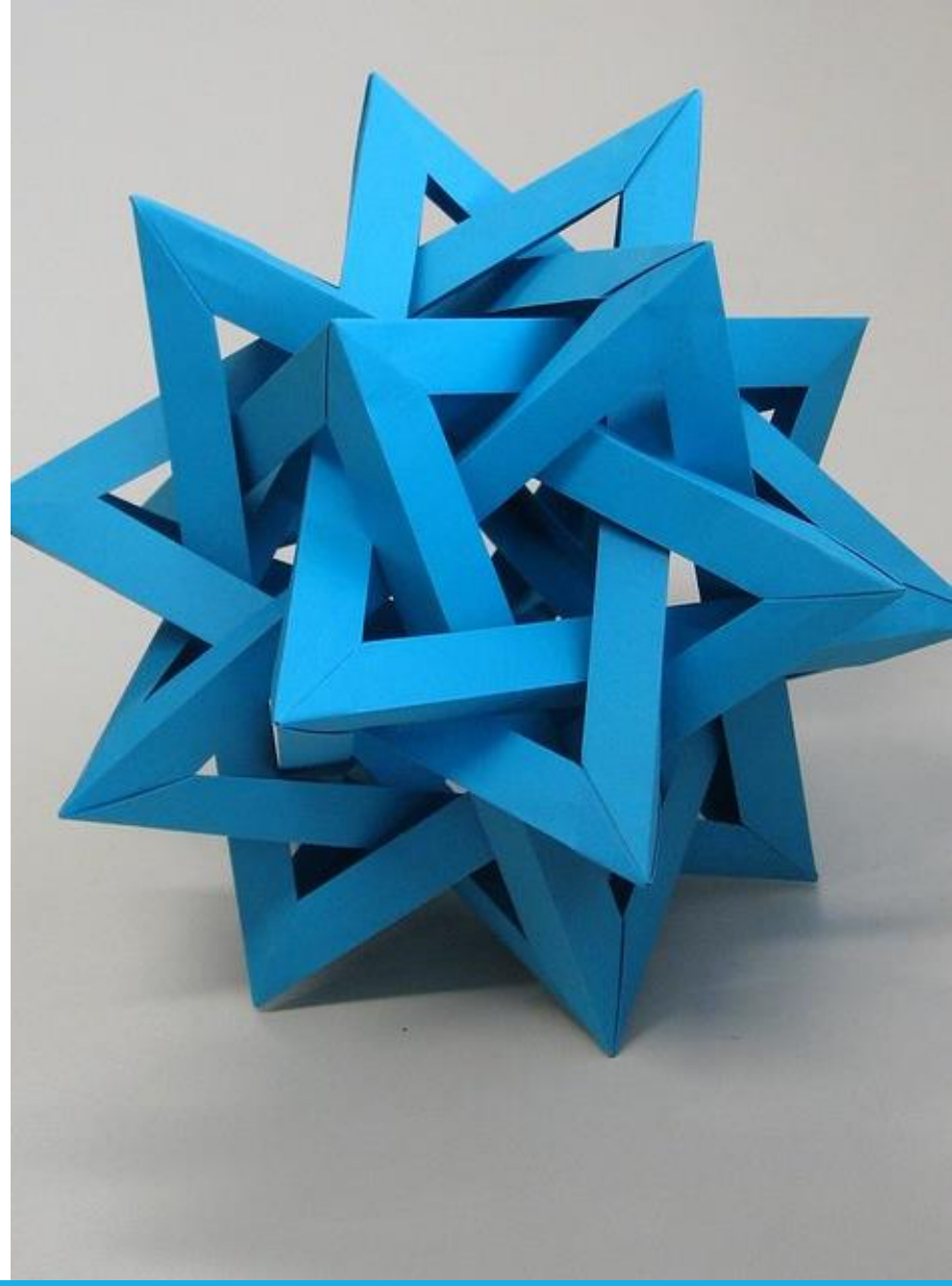
POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Unit P1: Introduction to programming

A SHORT INTRODUCTION TO HARDWARE, SOFTWARE, AND ALGORITHM DEVELOPMENT

Chapter 1



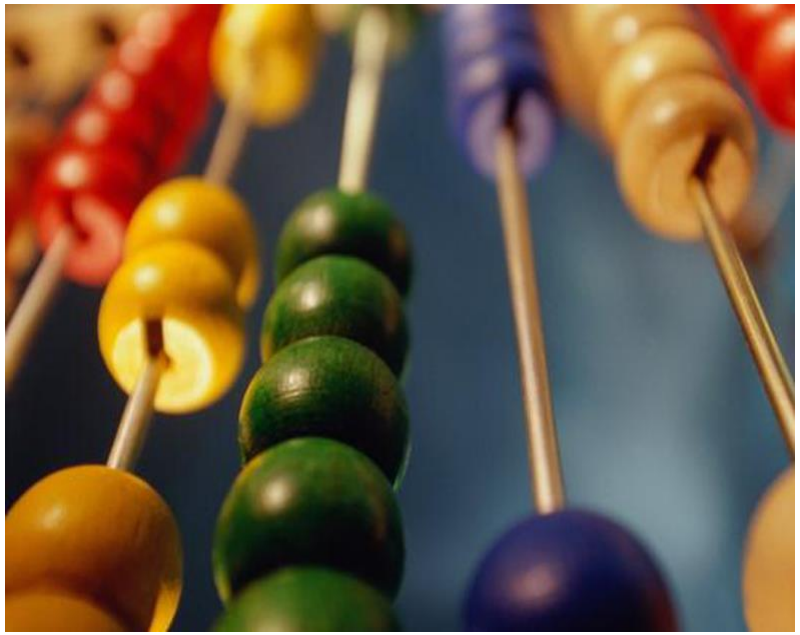
Unit P1: Goals

- Introduction to computers and programming
  - About computer hardware, software and programming
  - How to test, find and fix programming errors
  - How to use pseudocode to describe an algorithm
- Flow Charts as a support for problem solving
  - Representation
  - Design steps
- Introduction to Python
  - Tools
  - Language

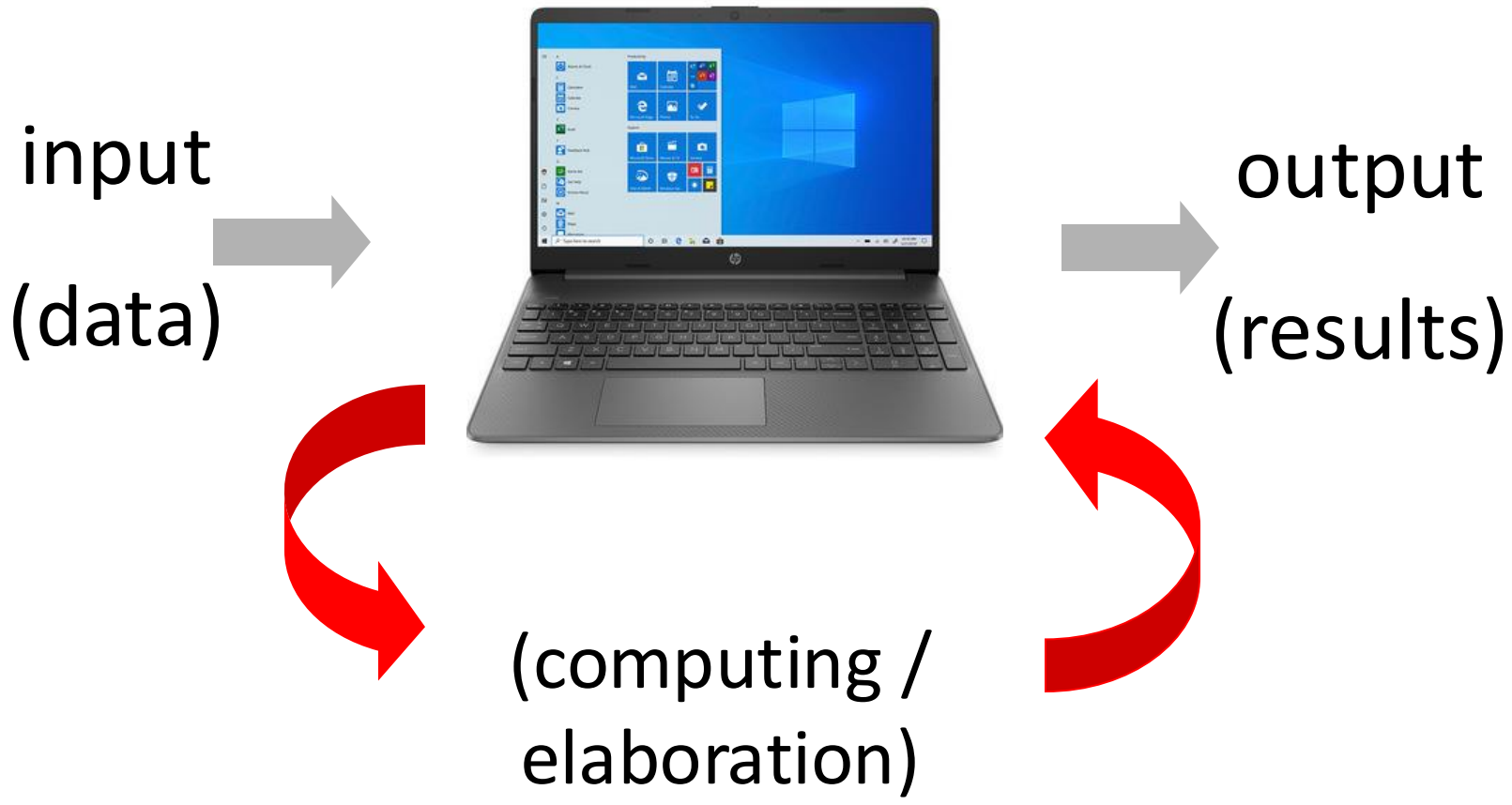
Computer Sciences Introduction

Definition of Computer Science

- Computer Science (Informatics) is the science devoted to the study of how to represent and manipulate **information**



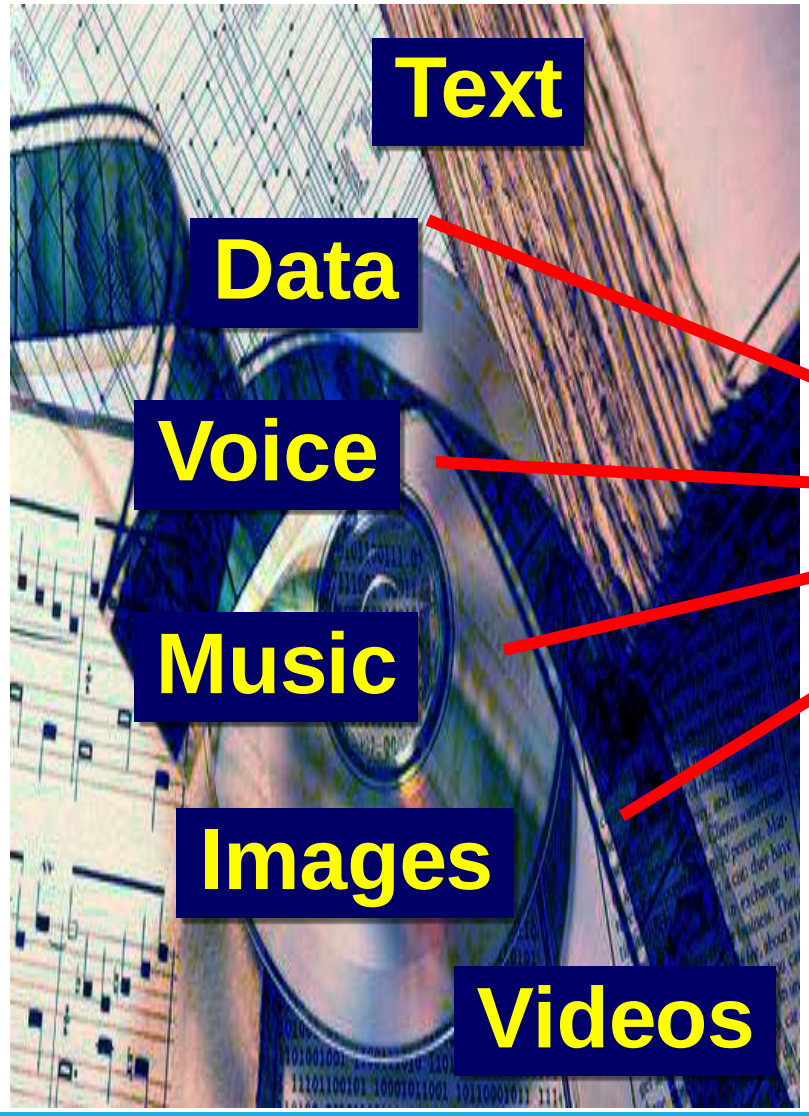
Electronic Computer



Problems

- How to **encode data** in a format that can be understood by the computer
- How to **encode the commands or instructions** into a sequence of operations that composes the desired elaboration
- How to **codify the results** in a format that can be understood by the human user

Digital information: everything becomes “bits”



01101100

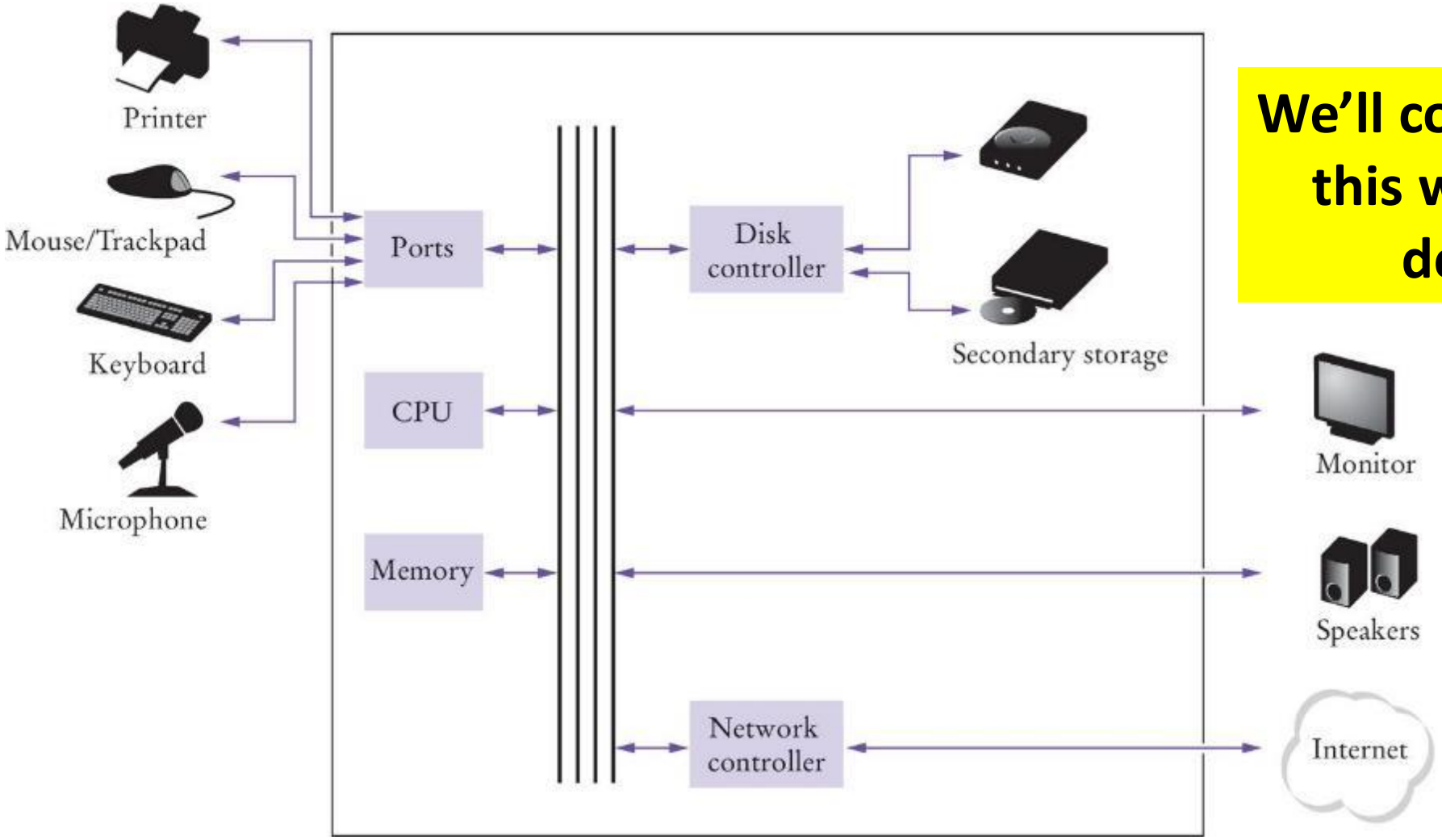
Hardware and Software

- An electronic computer is composed of two parts:
  - **Hardware:** Physical component, consisting of electronic devices, and mechanical, magnetic and optical parts
  - **Software:** "intangible" component consisting of:
    - **Programs**: The "instructions" for the hardware
    - **Data**: Information upon which programs operate

Hardware

- **Hardware** consists of the physical elements in a computer system.
  - Examples: monitor, mouse, external storage, keyboard, ….
- The **central processing unit** (CPU) performs program control and data processing
- **Storage devices** include **main memory** (RAM) and **secondary storage**
  - Hard disks
  - Flash drives
  - CD/DVD drives
- **Input / output devices** allow the user to interact with the computer
  - Mouse, keyboard, printer, screen…

Simplified View of a Computer's Hardware



We'll come back to this with more details!

Software

- **Software** is typically developed in the form of a **“program”**
  - Microsoft Word is an example of software
  - Videogames are software
  - Operating systems and device drivers are also software

Application Program (App)

- **Software**
  - Software is a **sequence of instructions, implemented in some language and translated to a form that can be executed or run on the computer.**
  - It manipulates and transforms **data**
- **Hardware** executes very basic instructions in rapid succession
  - Example: add two numbers.

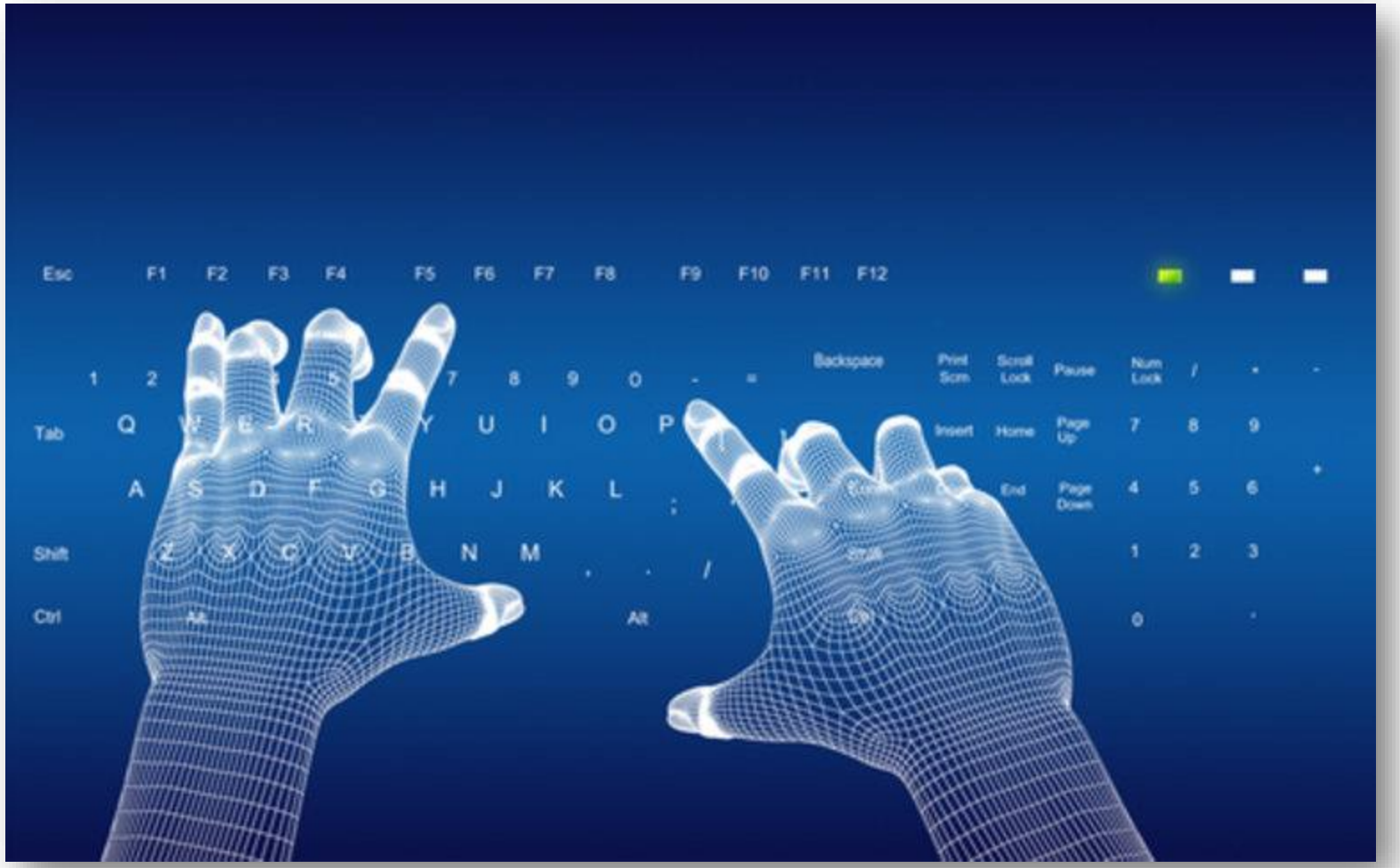
Computer Programs

- A computer program tells a computer the **sequence of steps** needed to complete a specific task
  - The program consists of a (very) large number of primitive (simple) instructions
- Computers can carry out a wide range of tasks because they can **execute** different programs
  - Each program is designed to direct the computer to work on a specific task
- **Programming:**
  - The act (and the **art**) of designing, implementing, and testing computer programs

Executing a Program

- Program instructions and data (such as text, numbers, audio, or video) are **stored in digital format (→ as bits)**
- To execute a program, it must be **brought (loaded) into memory**, where the CPU can read it.
- Then, the CPU executes the program **one instruction at a time**.
  - The program may react to input from the user.
- The **sequence of instructions** and the user input determine the program execution
  - The CPU reads data (including user input), modifies it, and writes it back to memory, the screen, or secondary storage (e.g. hard disk).

Programming



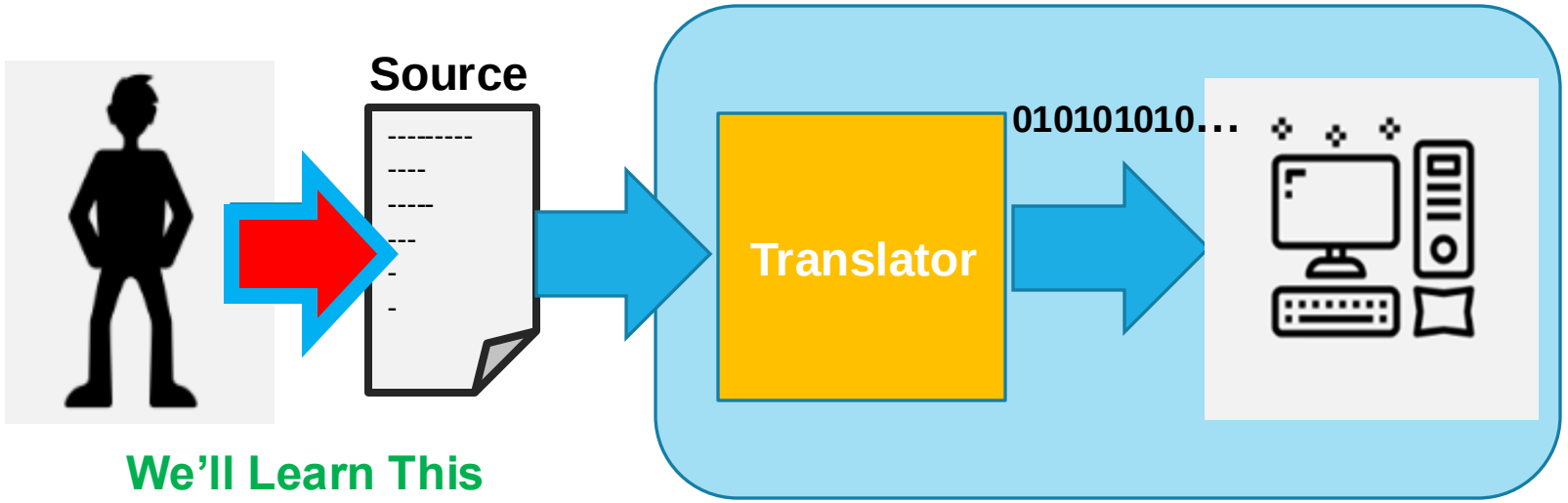
What does ”programming” mean?

- **Programming = designing the solution to a problem in a way that can be solved by a computer**

- To better understand the challenges of programming, it is important to understand the process of **building a program**

Building a Program

1. **Writing a Program**
   - **”Source” code**
   - Written in some programming language
2. **Translation** of the program **in a format that the computer can understand**
   - **”Executable” code**
   - Done automatically by a program called generically **translator**.



What does “Programming” mean?

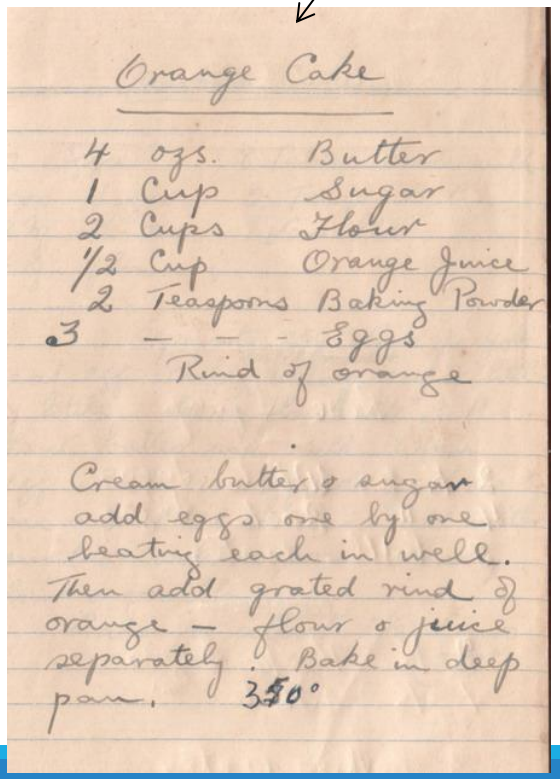
- Programming consists in writing a «document» (**source file**) which **describes the solution** to the considered problem in terms of a **sequence of instructions operating on data**

- In general, “**the**” solution to a problem does **not** exist
  - Programming consists of finding the *most efficient and effective solution* (according to appropriate metrics) for the problem

Cooking vs Programming

Hardware

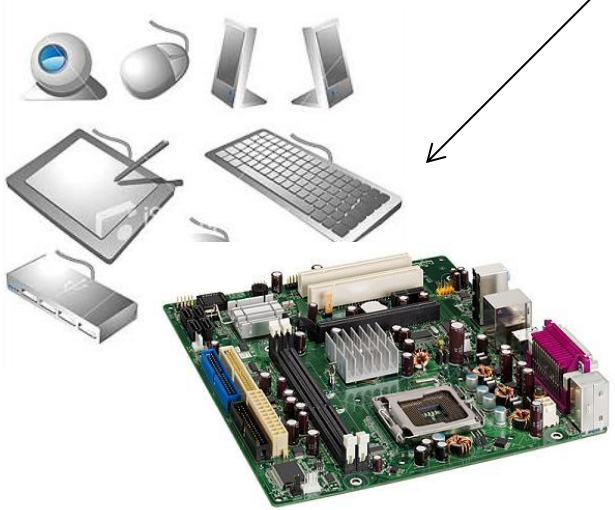
Software



Cooking vs Programming

Hardware

Software

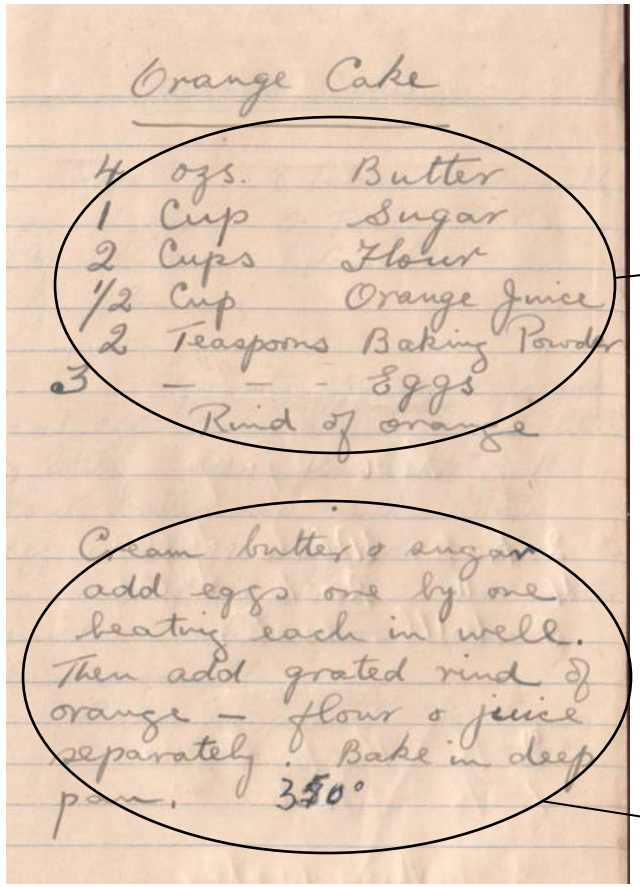


```
void CruiseControl_init(_C_CruiseControl * _C_)
{
    CruiseSpeedMgt_init(&(_C_->_C0_CruiseSpeedMgt));
    CruiseStateMgt_init(&(_C_->_C3_CruiseStateMgt));
    (_C_->_M_condact_0) = true;
    ThrottleCmd_init(&(_C_->_C4_ThrottleCmd));
    (_C_->_M_init) = true;
}

/* ========= */
/* MAIN NODE */
/* ========= */

void CruiseControl(_C_CruiseControl * _C_)
{
    bool BrakePressed;
    bool AcceleratorPressed;
    bool SpeedOutOffLimits;
    bool _L19;
/*#code for node CruiseControl */
/* call to node not expanded DetectPedalsPressed */
    (_C_->_Cn_DetectPedalsPressed._I0_Brake) = (_C_->_I
    (_C_->_Cn_DetectPedalsPressed._I1_Accelerator) = (_
    DetectPedalsPressed(&(_C_->_Cn_DetectPedalsPressed)
    BrakePressed = (_C_->_Cn_DetectPedalsPressed._O0_Br
    AcceleratorPressed =
        (_C_->_Cn_DetectPedalsPressed._O1_AcceleratorPr
/* call to node not expanded DetectSpeedLimits */
    (_C_->_Cn_DetectSpeedLimits._I0_speed) = (_C_->_I8_
    DetectSpeedLimits(&(_C_->_Cn_DetectSpeedLimits));
    SpeedOutOffLimits = (_C_->_Cn_DetectSpeedLimits._O0_
/* call to node not expanded CruiseStateMgt */
    (_C_->_C3_CruiseStateMgt._I0_BrakePressed) = BrakeP
```

Cooking vs Programming



Ingredients/Data

```
/* ========= */
/* MAIN NODE */
/* ========= */

void CruiseControl(_C_CruiseControl * _C_)
{
    bool BrakePressed;
    bool AcceleratorPressed;
    bool SpeedOutOffLimits;
    bool _L19;
/*#code for node CruiseControl */
/* call to node not expanded DetectPedalsPressed */
    (_C_->_Cn_DetectPedalsPressed._I0_Brake) = (_C_->_I
    (_C_->_Cn_DetectPedalsPressed._I1_Accelerator) = (_
    DetectPedalsPressed(&(_C_->_Cn_DetectPedalsPressed))
    BrakePressed = (_C_->_Cn_DetectPedalsPressed._O0_Br
    AcceleratorPressed =
        (_C_->_Cn_DetectPedalsPressed._O1_AcceleratorPr
/* call to node not expanded DetectSpeedLimits */
    (_C_->_Cn_DetectSpeedLimits._I0_speed) = (_C_->_I8_
    DetectSpeedLimits(&(_C_->_Cn_DetectSpeedLimits));
    SpeedOutOffLimits = (_C_->_Cn_DetectSpeedLimits._O0_
/* call to node not expanded CruiseStateMgt */
    (_C_->_C8_CruiseStateMgt._I0_BrakePressed) = BrakeP
```

Instructions/Code

What is “programming”?

- Programming is a “**creative**” task!
  - Every problem is **different** from every other problem
  - No silver bullets (**universal solutions**)
  - (Almost) **no systematic/analytic** solutions

- Programming is a complex operation
  - A “direct” approach (from the problem directly to the final source code) is almost impossible
  - The problem needs to be decomposed in sub tasks
  - Recognize known “patterns” (don’t reinvent the wheel)
  - Usually, organized in several stages (refinements)

Algorithms

1.7

Our First Definition

- **Algorithm**
- An algorithm is a **step-by-step, formal description** of **how to solve a problem**
- A "systematic procedure that produces — in a finite number of steps — the answer to a question or the solution of a problem"
- A program is an implementation of an algorithm in a certain language.

The name derives from Al-Khwārizmī (c. 780 — c. 850), Persian mathematician and astronomer

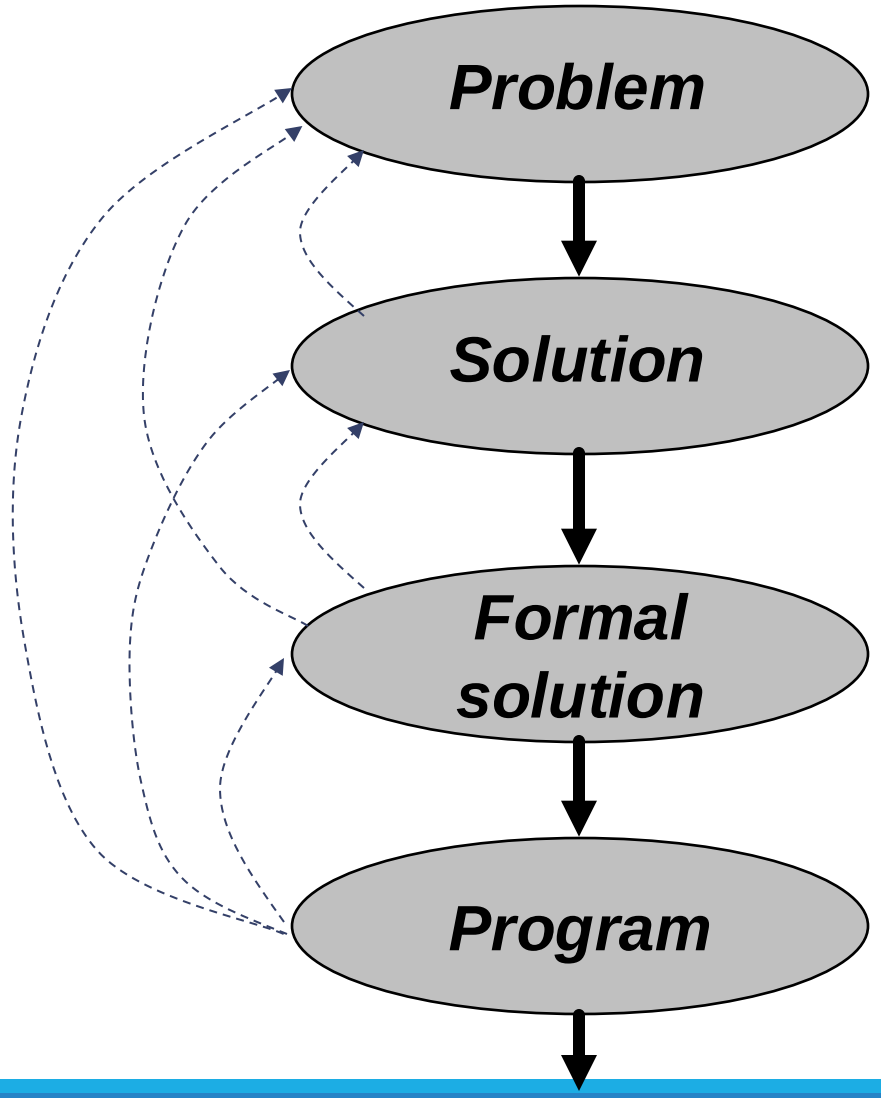
Find the biggest number

-9345

-1302

-5901

Designing a Program

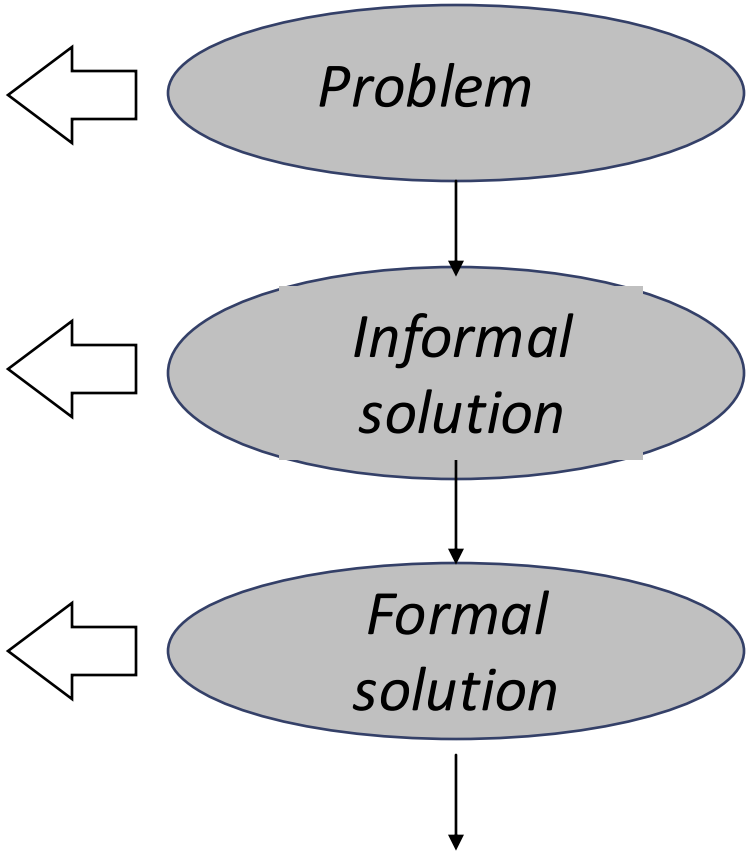


Algorithm!

Example of problem solving

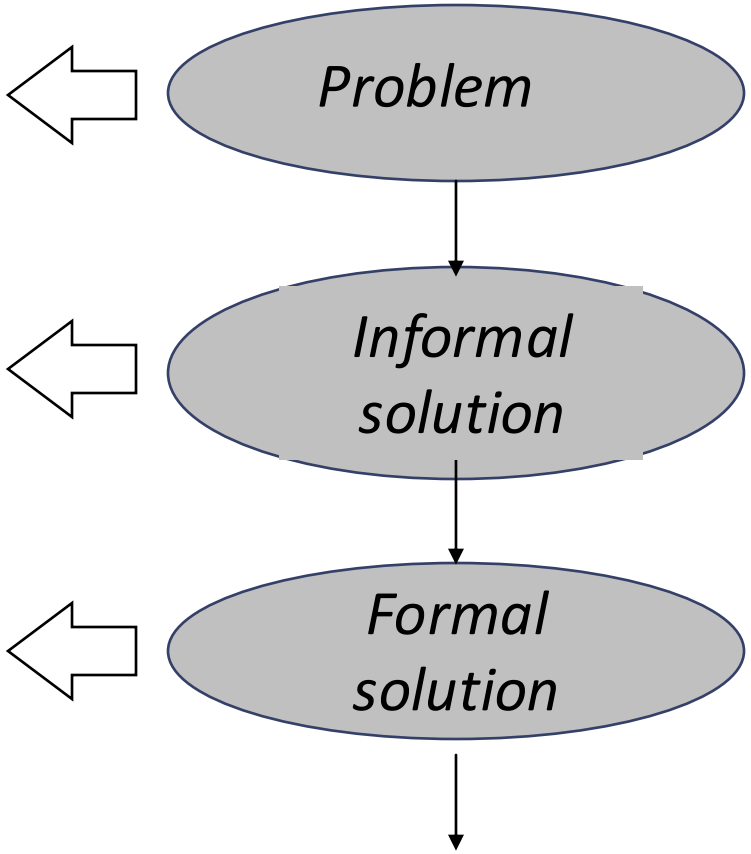
- Problem: Compute the maximum among A, B and C
- Solution: The maximum is the largest number between A, B and C…
- Formal solution:

  ??



Example of problem solving

- Problem: Compute the maximum among A, B and C
- Solution: The maximum is the largest number between A, B and C…
- Formal solution:
  1. if A >= B and A >= C then MAX = A;
  2. Else if B >= A and B >= C then MAX = B;
  3. Else if C >= A and C >= B then MAX = C;



Find the biggest number

```
1256 6765 6829 3847 3284 4798 3848
0185 3721 3689 4574 3985 4389 5743
4395 7239 8473 2854 3785 6648 0101
9840 7231 5672 1419 4385 6318 9910
3532 5498 6043 7542 8957 2149 3285
4903 2750 3019 2754 8018 3605 2389
2740 8993 7416 5415 9857 1624 8935
7318 7561 3746 5327 5432 5436 5078
5612 3465 7984 8764 1264 1547 8144
2194 1948 1282 5821 8975 4276 5219
```

Algorithms in everyday life

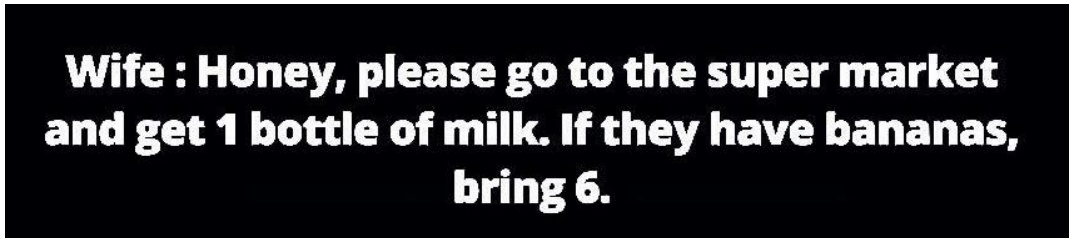
1. Pour some water
2. Light the fire
3. Wait
4. If water is not boiling, go back to n.3
5. Add pasta
6. Wait a little
7. Taste
8. If it's still raw go back to 6
9. Remove water

IS THERE AN ERROR IN THIS ALGORITHM?

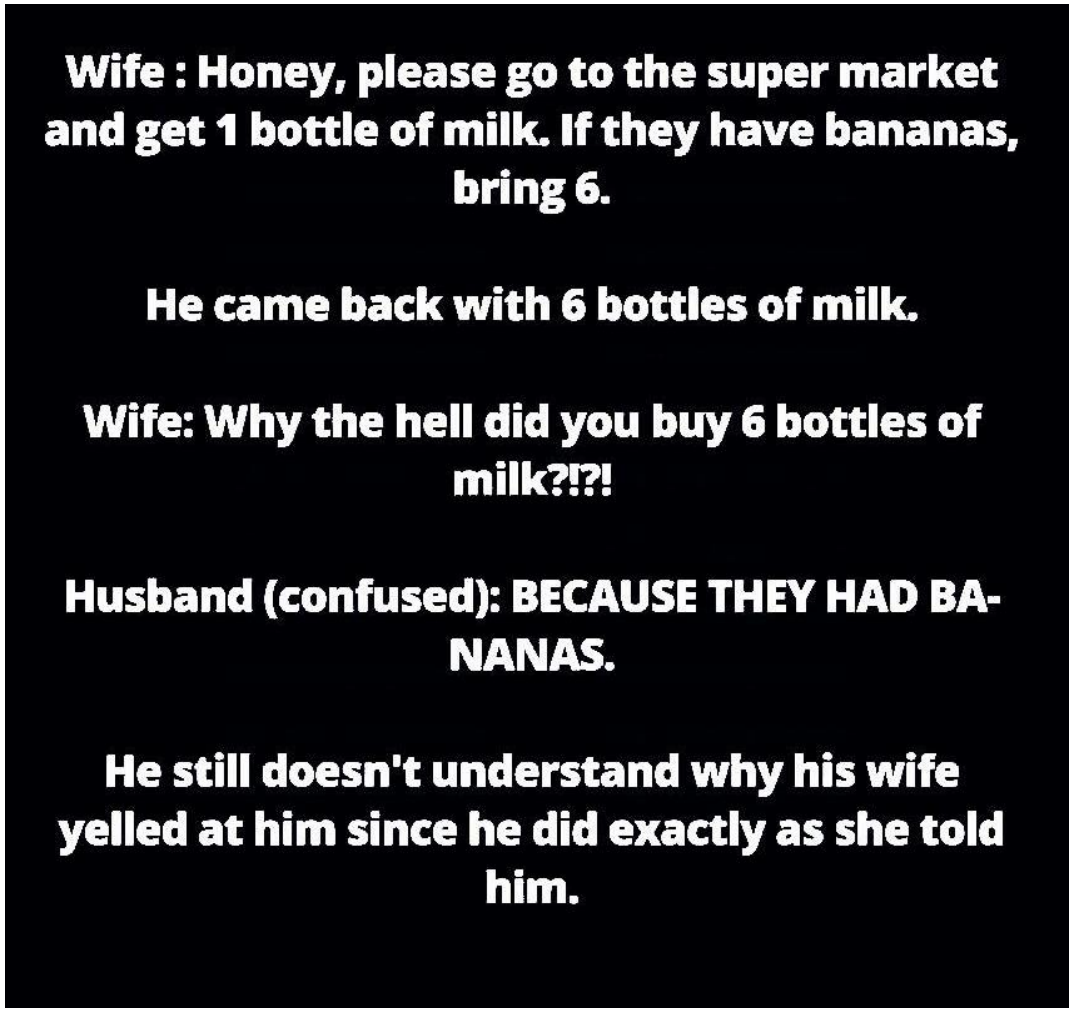
Algorithm: Formal Definition

- An algorithm describes a sequence of steps that is:
- **Unambiguous**
  - No “assumptions” are required to execute the algorithm
  - No “common sense” is assumed
  - The algorithm uses precise instructions
- **Simple**
  - Each step can be carried out in practice
- **Terminating**
  - The algorithm will eventually come to an end

Thinking Like a Programmer



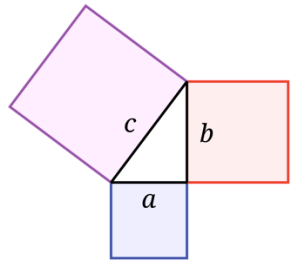
Thinking Like a Programmer



Ambiguous Algorithm!

Problem Solving: Algorithm Design

- We are already familiar with many algorithms
  - Calculate the area of a circle
  - Verify that a triangle is a right triangle
  - Solve a quadratic equation $ax^2 + bx + c = 0$



- Some problems are more complex and require to go through a variable number of steps
  - Find the **greatest common divisor** of two numbers

Example: Selecting a Car

- Problem Statement:
  - You have the choice of buying two cars
  - One is more fuel efficient than the other, but also more expensive
  - You know the **price and fuel efficiency** (in miles per gallon, mpg) of both cars
  - You know the price of fuel per gallon **($4.00 / gallon)**
  - You know that you drive **15000 miles per year**
  - You plan to **keep the car for ten years**
  - Which car is the better deal?

Developing the Algorithm

- Determine the inputs and outputs
- From the problem statement we know:
  - Car 1: Purchase price, Fuel Efficiency
  - Car 2: Purchase price, Fuel Efficiency
  - Price per gallon = $4.00
  - Annual miles driven= 15,000
  - Length of time = 10 years
- For each car we need to calculate:
  - Annual fuel consumed for each car
  - Operating cost for each car
  - Total cost of each Car
- Then we select the car with the lowest total cost

Formalizing the Algorithm

- Break down the problem into smaller tasks
  - 'Calculate total cost' for each car
  - To calculate the total cost for each year we need to calculate the operating cost
  - The operating cost depends on the annual fuel cost
  - The annual fuel cost is the price per gallon times the annual fuel consumed
  - The annual fuel consumed is the annual miles drive divided by fuel efficiency
- Describe each subtask
  - total cost = purchase price + operating cost
  - etc.

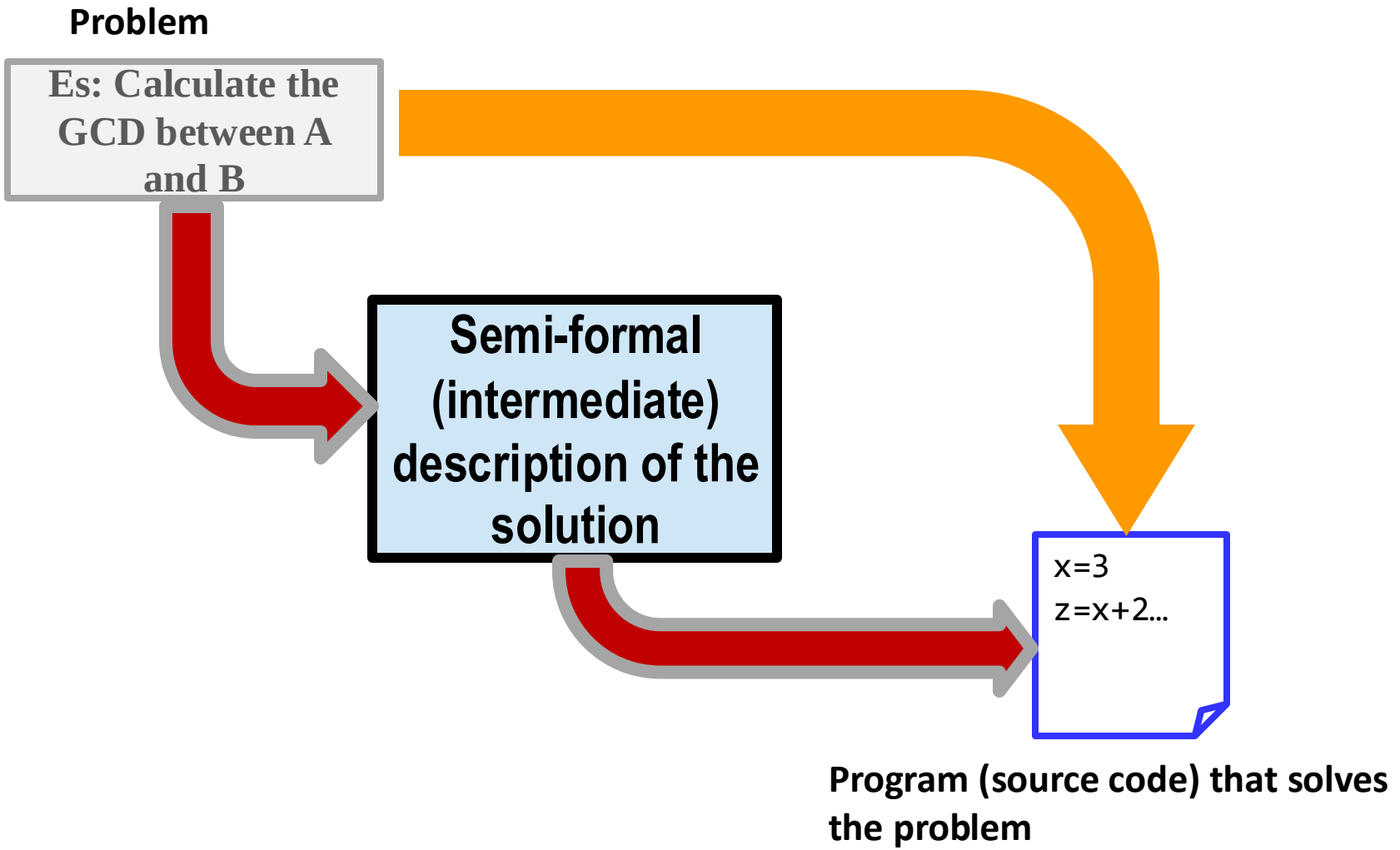
Intermediate Step

- The program is the **final representation of an algorithm**/solution

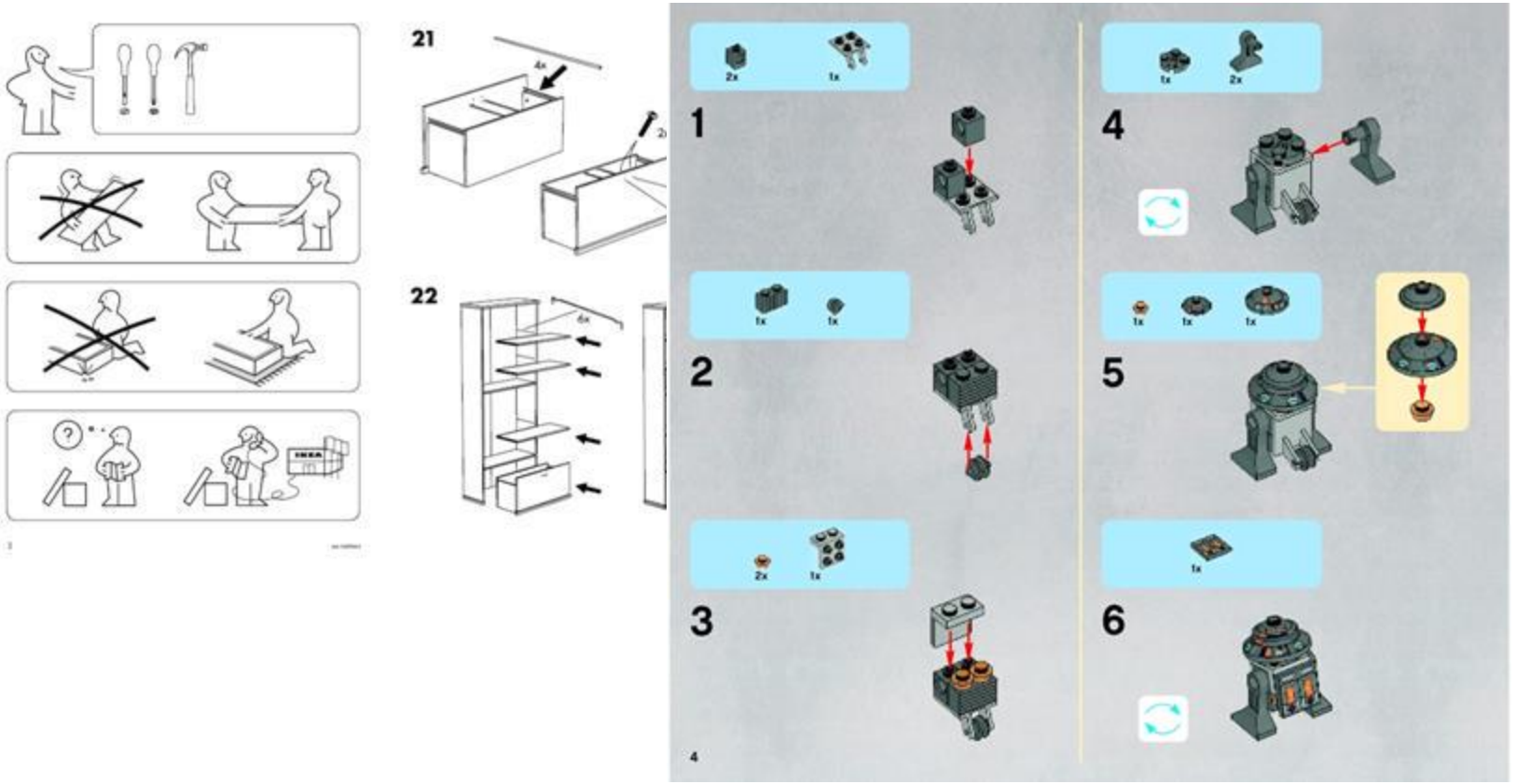
- Except in very simple cases, a direct approach (from the problem to the source program) is not feasible

- So let's decouple problem of "**finding a solution**" from the problem of "**writing the program**", using a **"semi-formal" intermediate solution**

From Problem to Program



Intermediate Representation: Inspiration



- Structured visual language
- Sequential operations
- Sub-operations
- Repetitions

Intermediate Representation of a Solution

- For a computer algorithm, we need a formalism that is:
  - Unambiguous
  - Aware of the operations that the computer can perform
  - Easy to interpret

Intermediate Representation of a Solution

Two main options:

- **Pseudo-code**
  - Half-way between natural language and a programming language
- **Flow Charts**
  - Intuitive graphical formalism
  - …Struggle to represent complex or more abstract operations

Possible Pseudo-code for the “car problem”

- For each Car, compute the total cost
  - annual fuel consumed = annual miles driven / fuel efficiency
  - annual fuel cost = price per gallon * annual fuel consumed
  - operating cost = length of time * annual fuel cost
  - total cost = purchase price + operating cost
- If total cost1 < total cost2
  - Choose Car1
- Else
  - Choose Car2

Bank Account Example

- You put $10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original?

Bank Account Example

- How would you solve it?
  - Make a table
  - Add lines until done
- Use a spreadsheet!

year	balance
0	10000
1	10000.00 x 1.05 = 10500.00
2	10500.00 x 1.05 = 11025.00
3	11025.00 x 1.05 = 11576.25
4	11576.25 x 1.05 = 12155.06

Developing the Algorithm

- You put $10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original?

year	balance
0	10000

- Break it into steps
  - Start with a year value of 0 and a balance of $10,000
  - Repeat the following while the balance is less than $20,000
    - Add 1 to the year value
    - Multiply the balance by 1.05
    - (5% increase)

year	balance
0	10000
1	10500
14	19799.32
15	20789.28

  - Report the final year value as the answer

Formalizing the Algorithm: Rules

- I must solve the problem, and imagine I have one sheet of paper (memory) and a pen, only

- Every information that I need to remember must be written on the paper

Formalized Algorithm and Pseudo-code

- Set the year value of 0
- Set the balance to $10,000
- While the balance is less than $20,000
  - Add 1 to the year value
  - Multiply the balance by 1.05
- Report the final year value as the answer

Pseudo-code: other examples

1

```
START
Write: "Enter a number n"
ACQUIRE n from user
If the remainder of the division by 2 is 0:
        Write: "The number is even"
Otherwise:
        Write: "The number is odd".
END
```

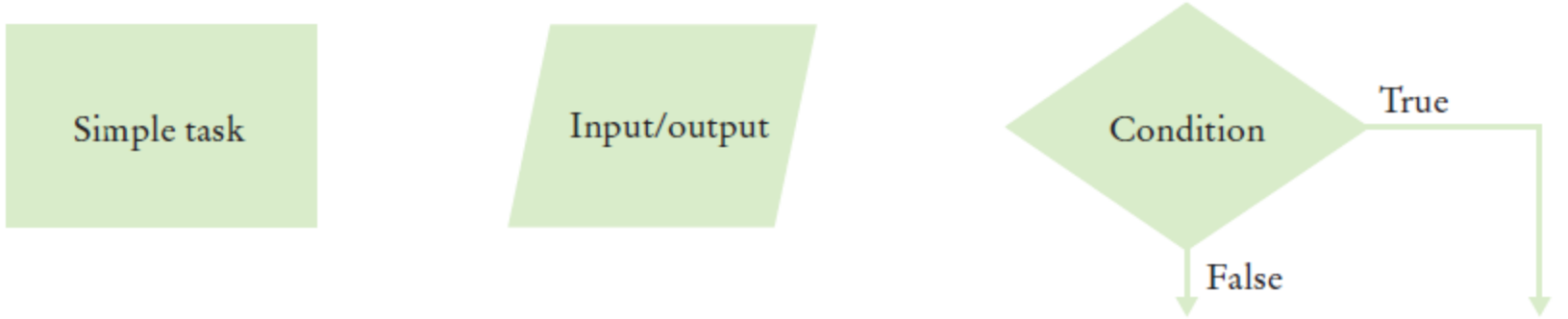
2

```
START
WRITE "Give me a value: "
ACQUIRE b
WRITE "Give me a second value: "
ACQUIRE e;
p= 1
IF (e<0):
     e = - e
     b = 1/b
REPEAT WHILE (e>0):
    p = p * b
    e = e – 1
WRITE "The result is"
WRITE p
END
```

Flowcharts

Problem Solving: Flowcharts

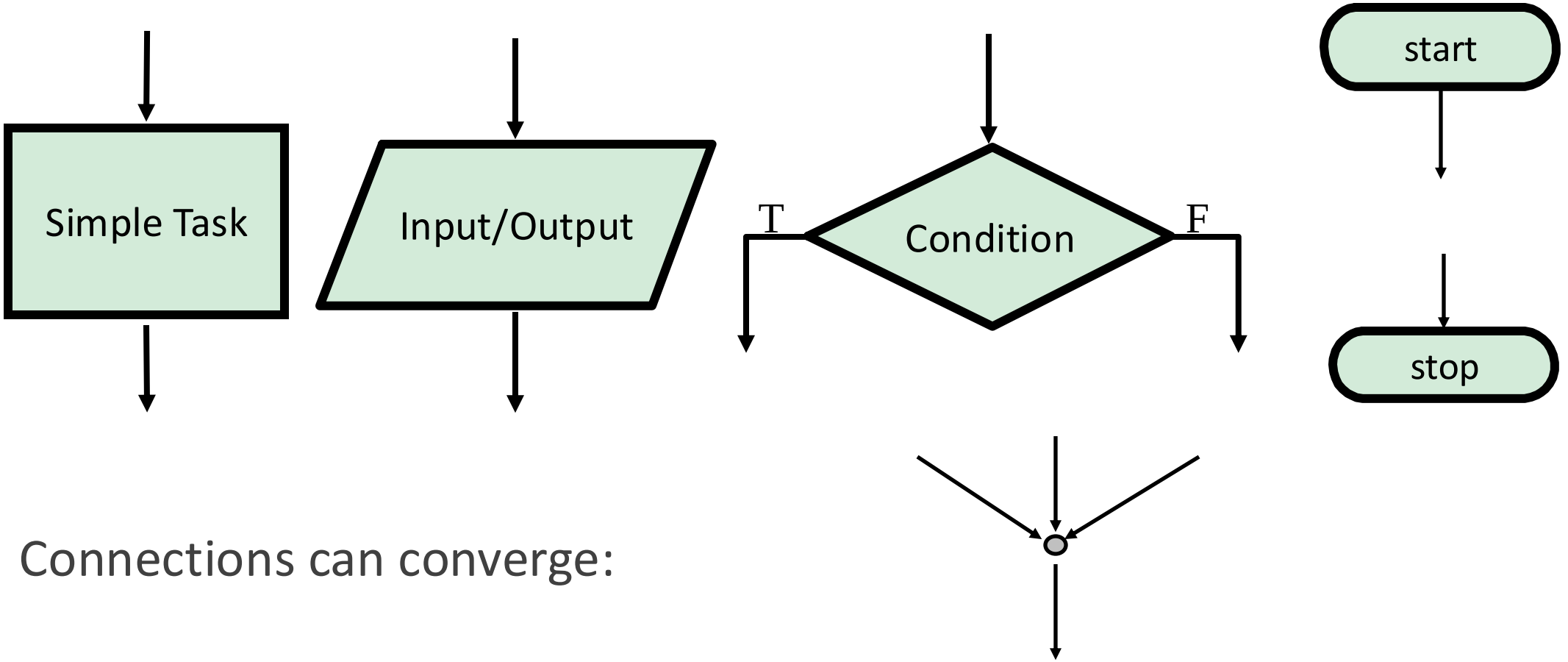
- A flowchart shows the structure of decisions and tasks to solve a problem
- Basic flowchart elements:



- Connect them with arrows
- Each branch of a decision can contain tasks and further decisions

Flow Charts

- Connection rules between blocks



- Connections can converge:

Using Flowcharts

- Flowcharts are an excellent tool
- They can help you visualize the flow of your algorithm
- Building the flowchart
  - Link your tasks and input / output boxes in the sequence they need to be executed
  - When you need to make a decision use the diamond (a conditional statement) with two outcomes
  - **Never point an arrow inside another branch**

Flowcharts: Abstraction Layer

- What can I write inside the blocks (especially action blocks)?

- More precisely, what level of complexity can the operations I write inside the blocks have?

Flowcharts: Abstraction Layer

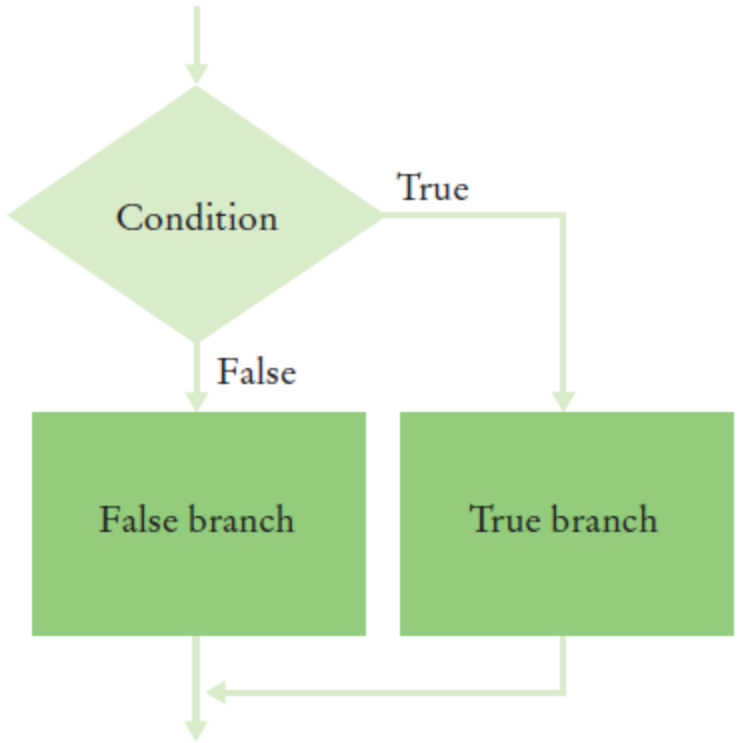
- Action blocks should contain actions corresponding to the **operations performed by a computer (microprocessor)**
  - Micro-operations
- Low level of abstraction

  - **Arithmetic operations between numeric values**
  - **Moving data**
  - **Acquisition (input) and printing (output)**
  - **Comparison of numerical data**
  - **Logical operations between logical conditions (union, intersection)**

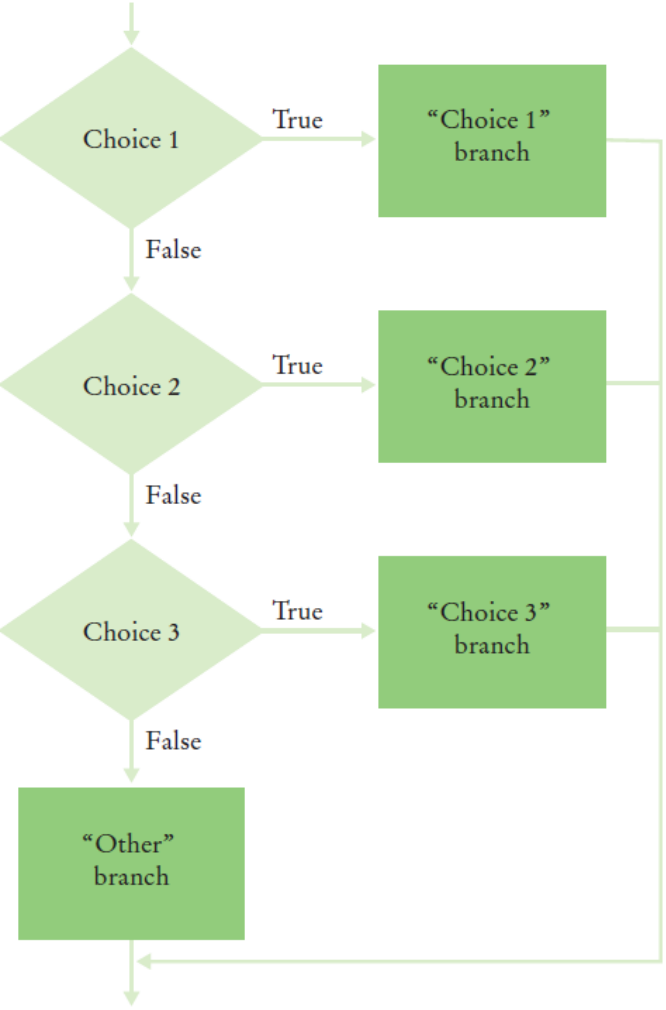
- These are the operations that the computer circuits actually implement (as we will see later)

Conditional Flowcharts

- Two Outcomes

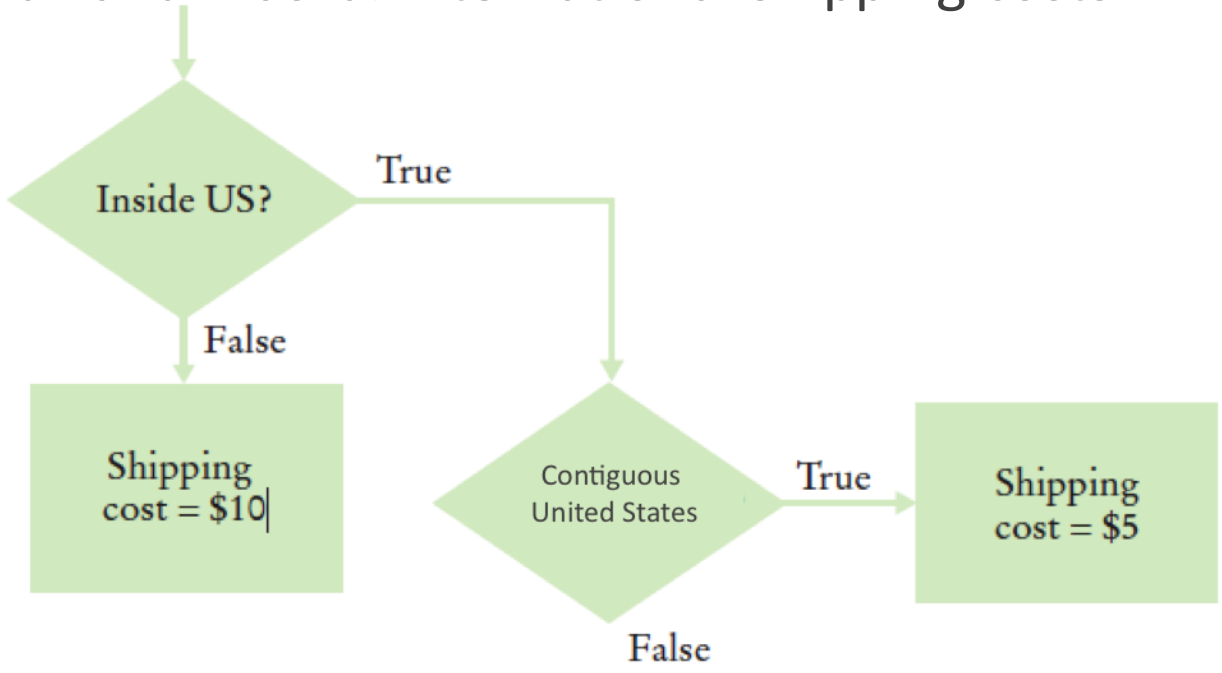


- Multiple Outcomes



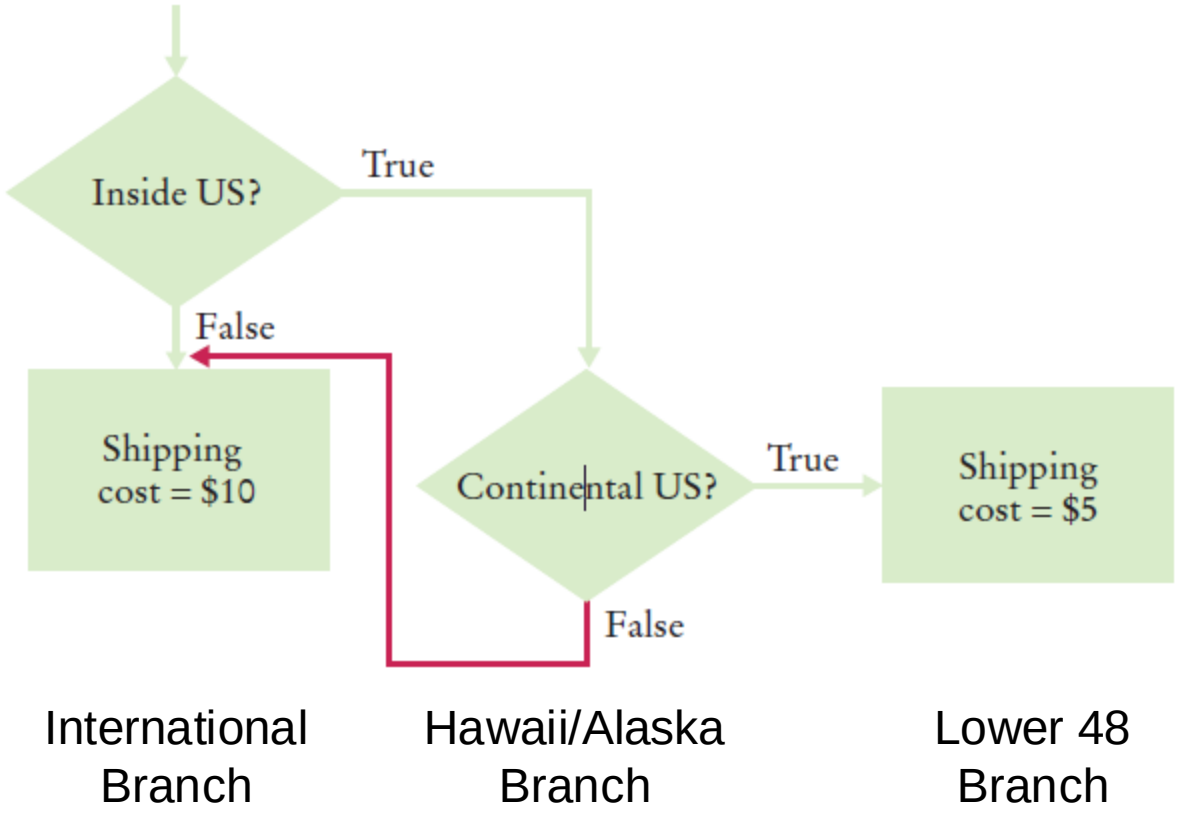
Shipping Cost flowchart

- **Problem: we want to compute how much we will pay to ship a packet to a friend**
- Available data: Shipping costs are $5 inside the contiguous United States (Lower 48 states), and $10 to Hawaii and Alaska. International shipping costs are also $10.
- Three Branches:



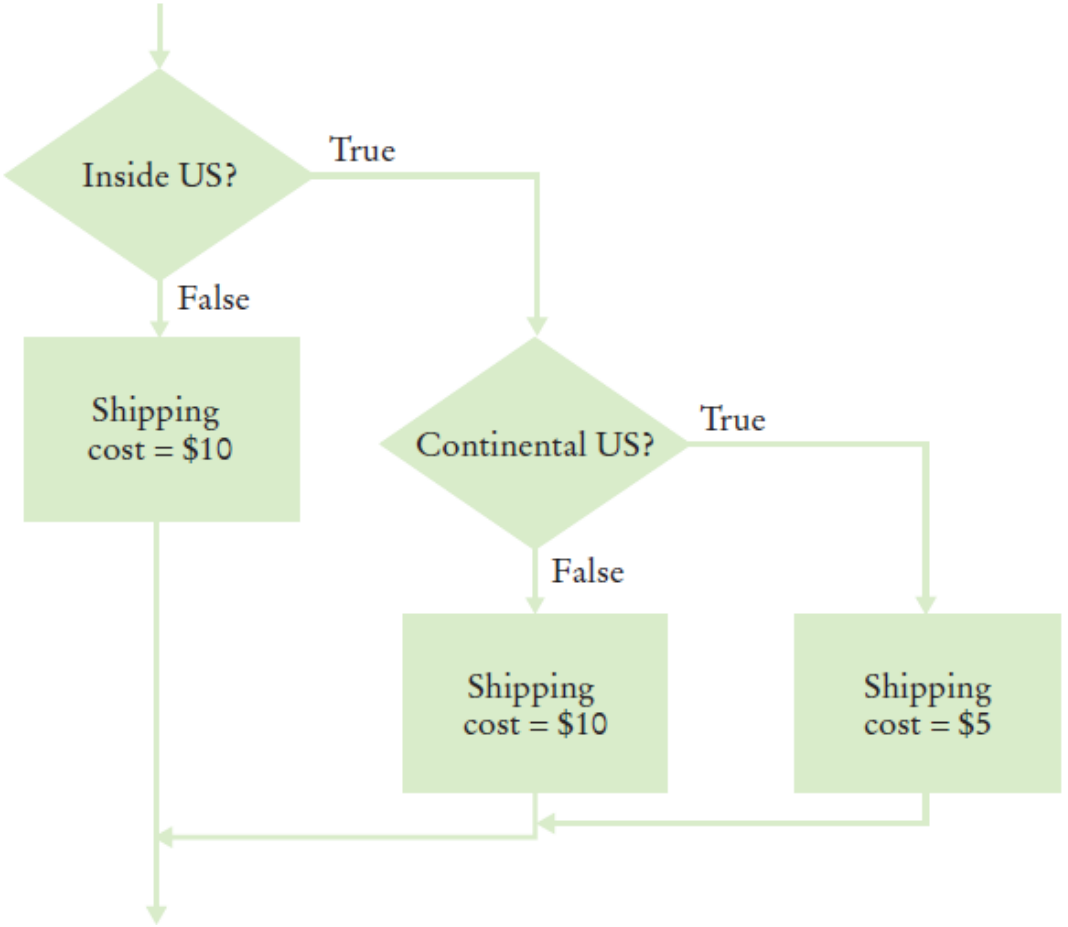
Don’t Connect Branches!

- Shipping costs are $5 inside the United States, except that to Hawaii and Alaska they are $10. International shipping costs are also $10.
- Don’t do this!



Shipping Cost Flowchart

- Shipping costs are $5 inside the United States, except that to Hawaii and Alaska they are $10. International shipping costs are also $10.
- **Do this!**

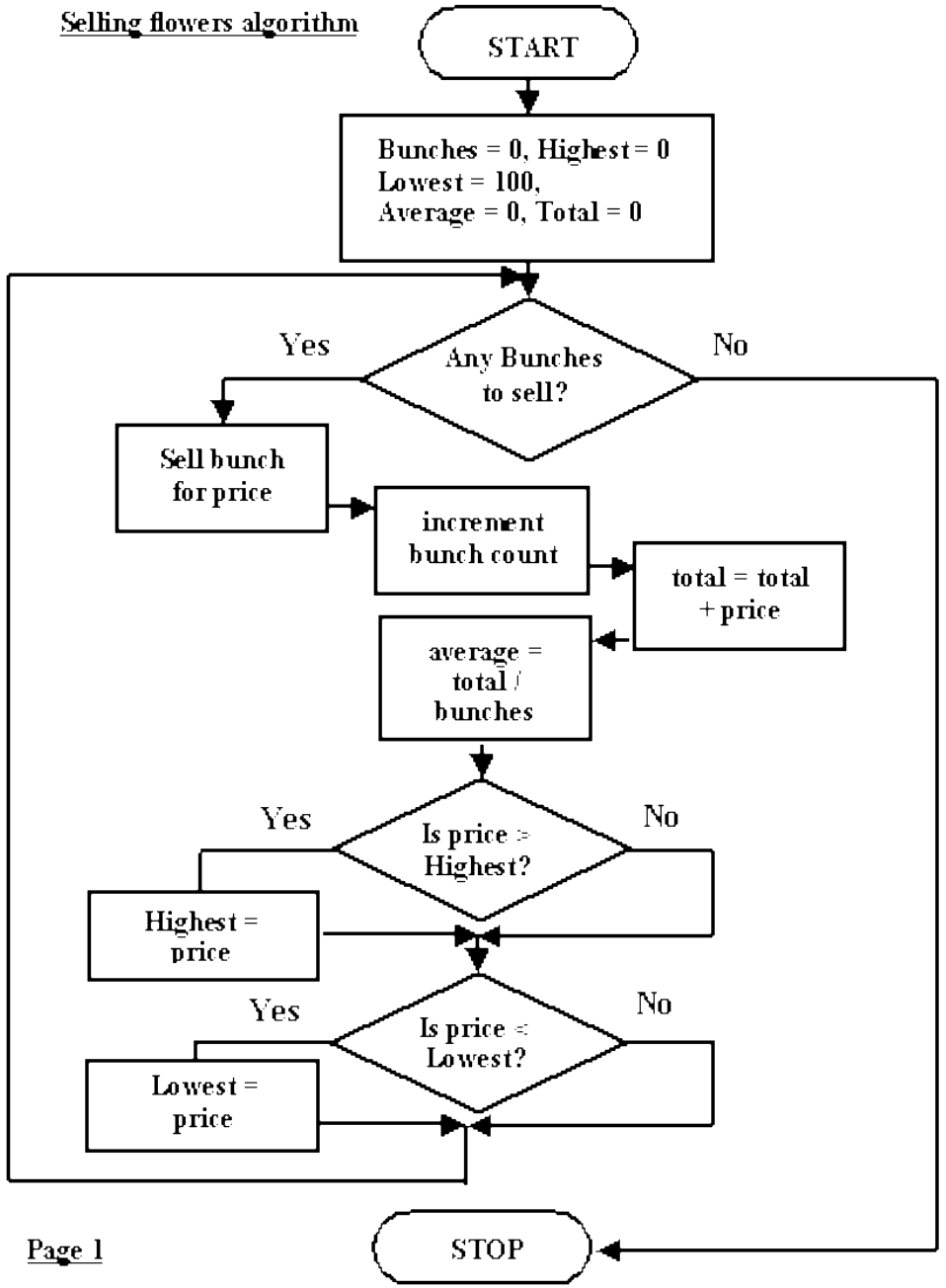


Exercise

- Fred sells bunches of flowers at the local shopping center.

- One day Fred's boss, Joe, tells Fred that at any time during the day he (Joe) will need to know:
  - how many bunches of flowers have been sold
  - what was the value of the most expensive bunch sold
  - what was the value of the least expensive bunch sold
  - what is the average value of bunches sold

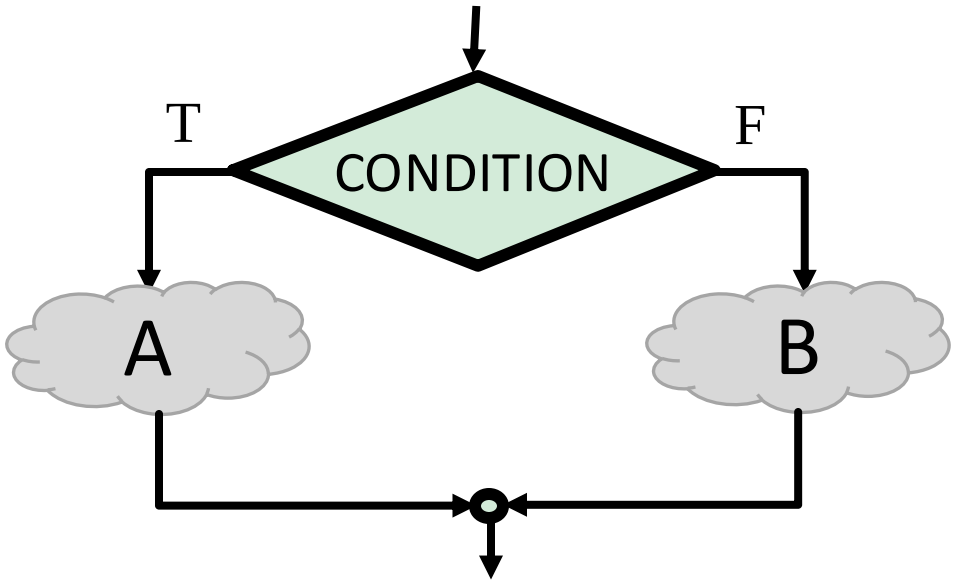
Exercise


Selling flowers algorithm
START
Bunches = 0, Highest = 0
Lowest = 100,
Average = 0, Total = 0
Yes
Any Bunches to sell?
No
Sell bunch for price
increment bunch count
total = total + price
average = total / bunches
Yes
Is price > Highest?
No
Highest = price
Yes
Is price < Lowest?
No
Lowest = price
Page 1
STOP

Try to think at a higher level

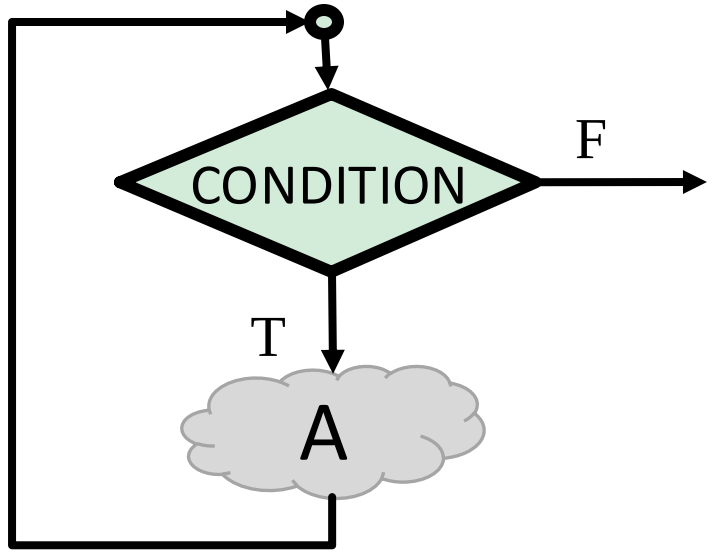
- Don’t think just in terms of **SINGLE** blocks, but also in terms of **multi-block structures**
  - Two fundamental “structures”

DECISIONS



“If CONDITION is true do A, else do B"

ITERATION (LOOP)



“While CONDITION is true do A”

Flowchart Exercises

GCD(A,B)

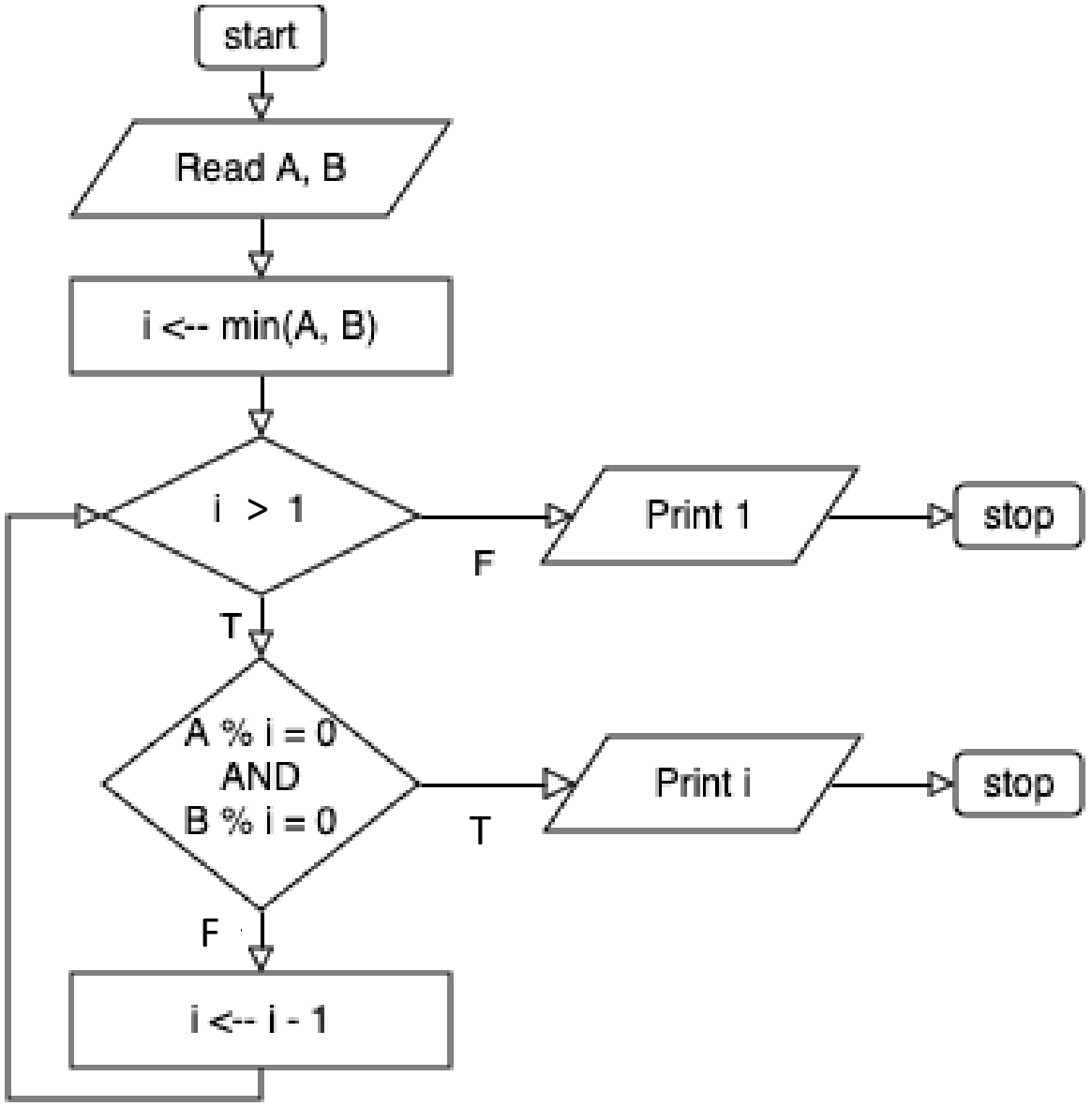
- Let's build a simple algorithm to find the GCD between two numbers
- Informal description

  For each value ***i*** between ***min(A,B)*** and 2 (in decreasing order), verify if both ***A*** and ***B*** are divisible by ***i.***

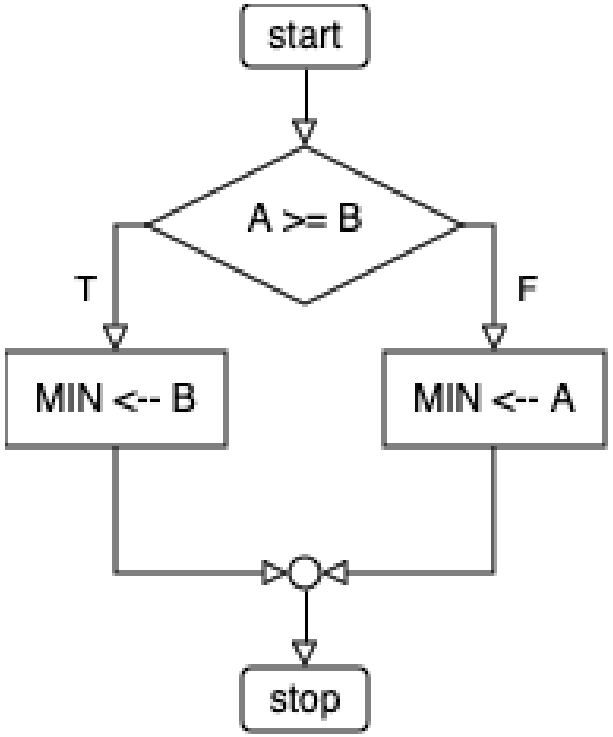
  If both conditions are true, then i is the GCD(A,B) and we can terminate

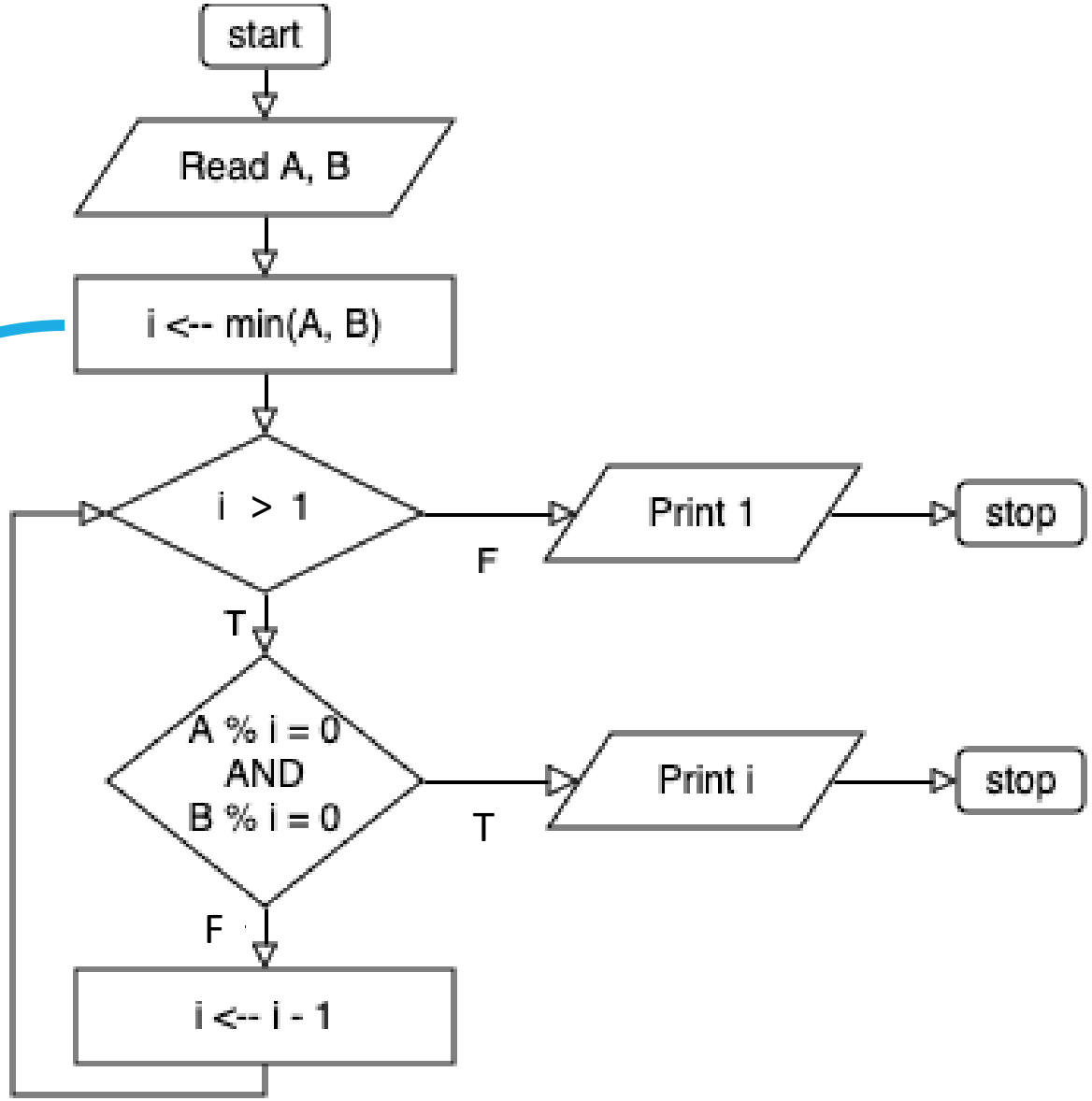
- Let's turn it into a flowchart.

GCD (A, B)


start
Read A, B
i <-- min(A, B)
i > 1
F
Print 1
stop
T
A % i = 0
AND
B % i = 0
T
Print i
stop
F
i <-- i - 1

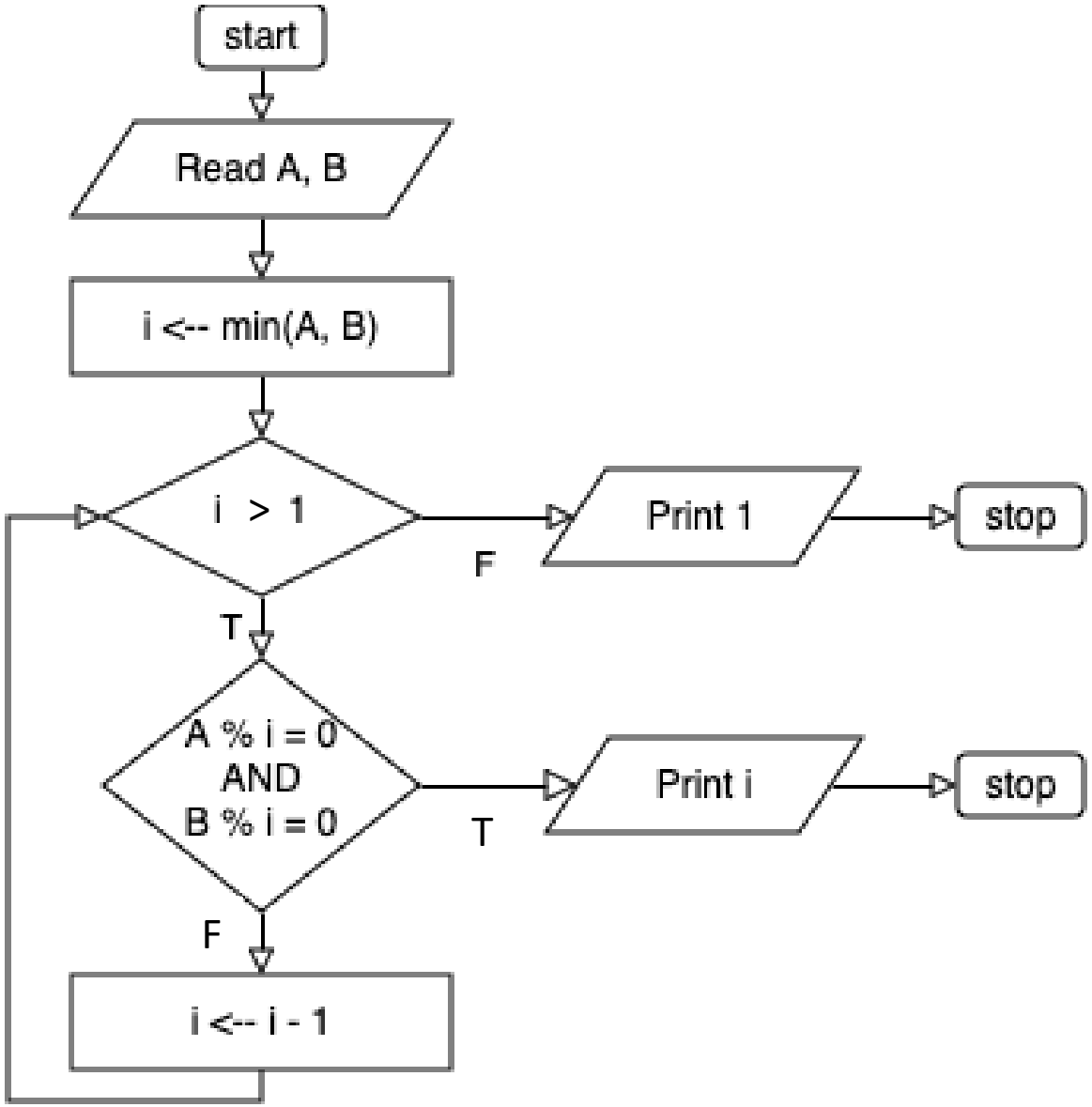
GCD (A, B)


start
A >= B
T
F
MIN <-- B
MIN <-- A
stop


start
Read A, B
i <-- min(A, B)
i > 1
F
Print 1
stop
T
A % i = 0
AND
B % i = 0
T
Print i
stop
F
i <-- i - 1

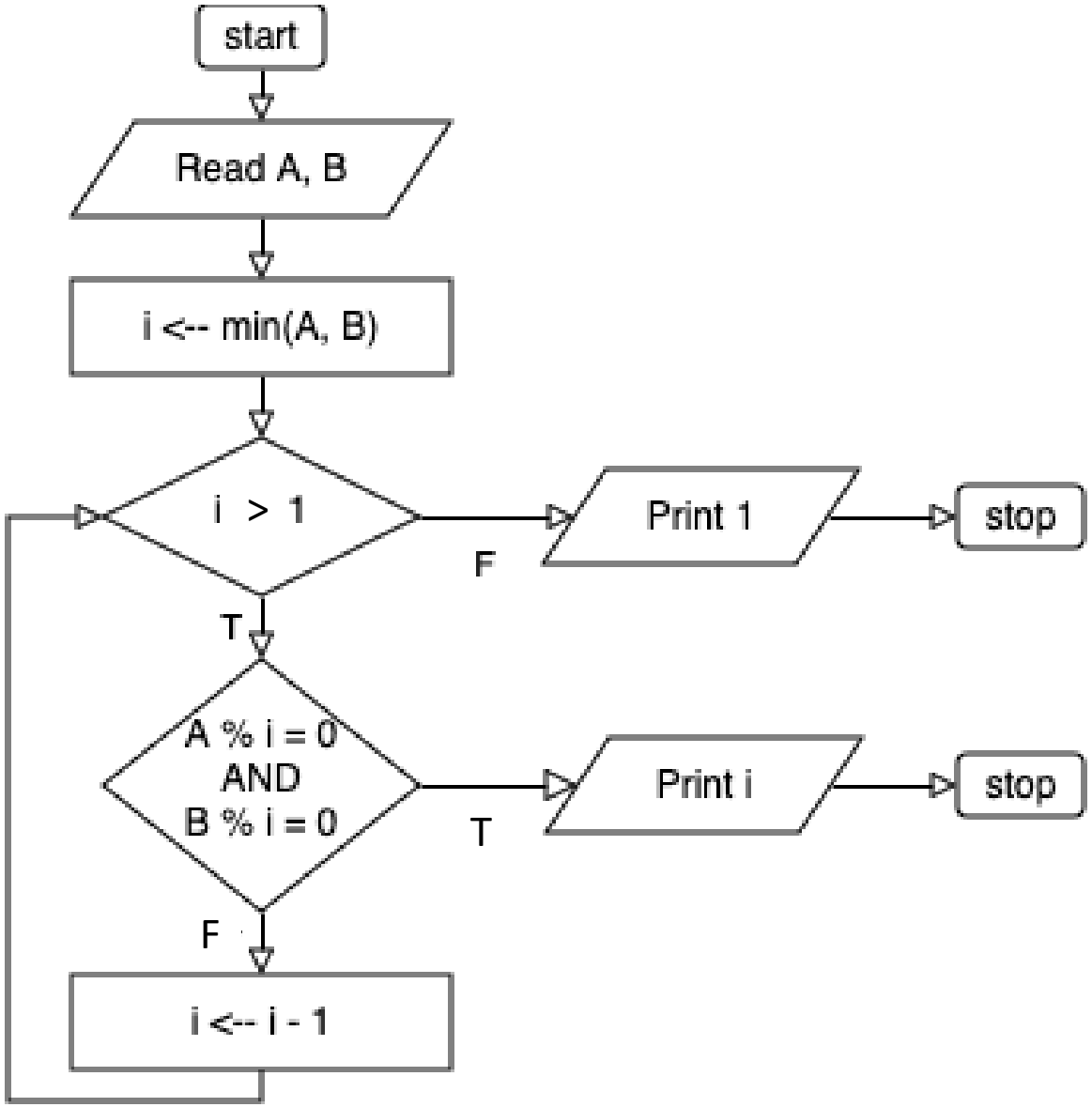
GCD (A, B)

- Example: **A=12, B=8**
- **Initialization: i ← min(A, B) = 8**
- **i >= 1? TRUE**
- **i divides A? FALSE | i divides B? TRUE**
- **i <-- 7**
- **i >= 1? TRUE**
- **i divides A? FALSE | i divides B? FALSE**
- **i <-- 6**
- **(…continues…)**



GCD (A, B)

- Example: **A=12, B=8**
- **(…continues…)**
- **i >= 1? TRUE**
- **i divides A? TRUE | i divides B? FALSE**
- **i <-- 5**
- **i >= 1? TRUE**
- **i divides A? FALSE | i divides B? FALSE**
- **i <-- 4**
- **i >= 1? TRUE**
- **i divides A? TRUE | i divides B? TRUE**
- **Print "i" → Print "4"**
- **Stop**



Exercise

- Given an integer number **N** entered by the user, determine if it is a prime number
- Let's use the definition: "**divisible only by itself and 1**"
- Said otherwise: "**not divisible by any number between 2 and N-1**"

Exercise

- Solution?

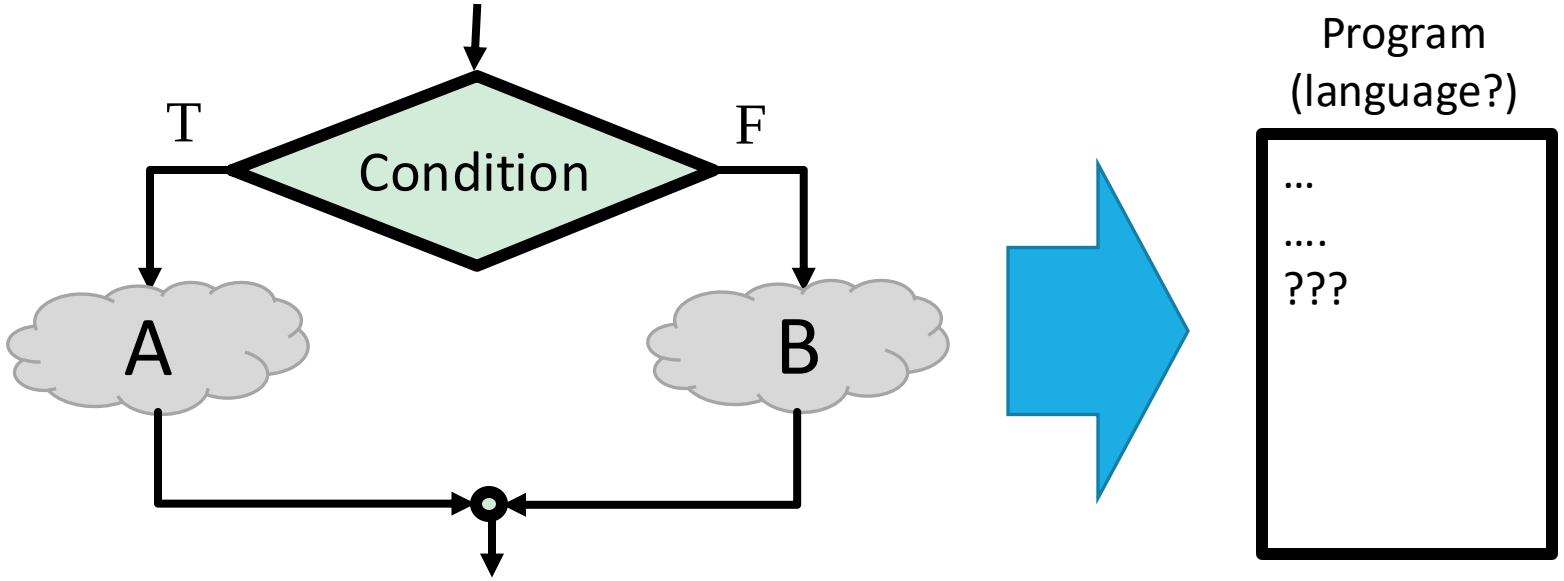
More Exercises

You will find a video lecture on the course website with some exercises on flowcharts.

Programming Languages

From Solution to Program

- Writing a program is almost “automatic”, if you start from a semi-formal solution (pseudo-code or flow chart)
- Different programming languages offer different constructs and instructions, of varying complexity



Which languages?

- Different levels of abstraction
  - **High-level languages**
    - Language statements have a complexity equivalent to **flow chart blocks**, such as assignments, conditions, cycles, …
    - Examples: C, C++, Java, JavaScript, Python, etc.
    - Independent from the hardware
  - **Assembly languages**
    - Language statements correspond to micro-architecture instructions
    - Highly dependent from the hardware
    - Example: Assembly language of the Intel Core microprocessor

Which languages? – Examples

- High Level Language

```
…
if x > 3:
  x = x + 1
…
```

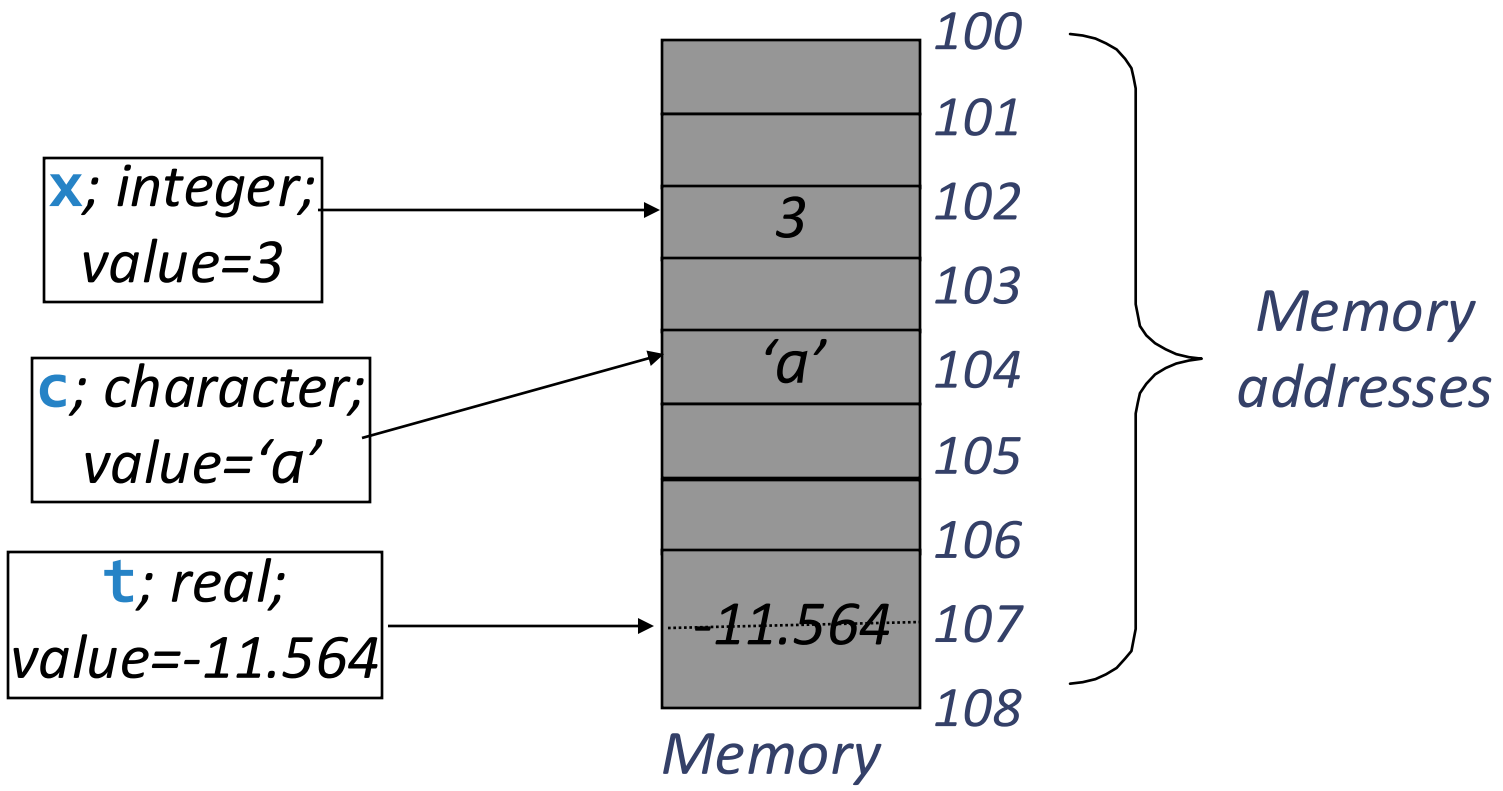
- Assembly Language

```
…
LOAD Reg1, Mem[1000]
ADD Reg1, 10
…
```

Specific for a given architecture (microprocessor)

Data abstraction

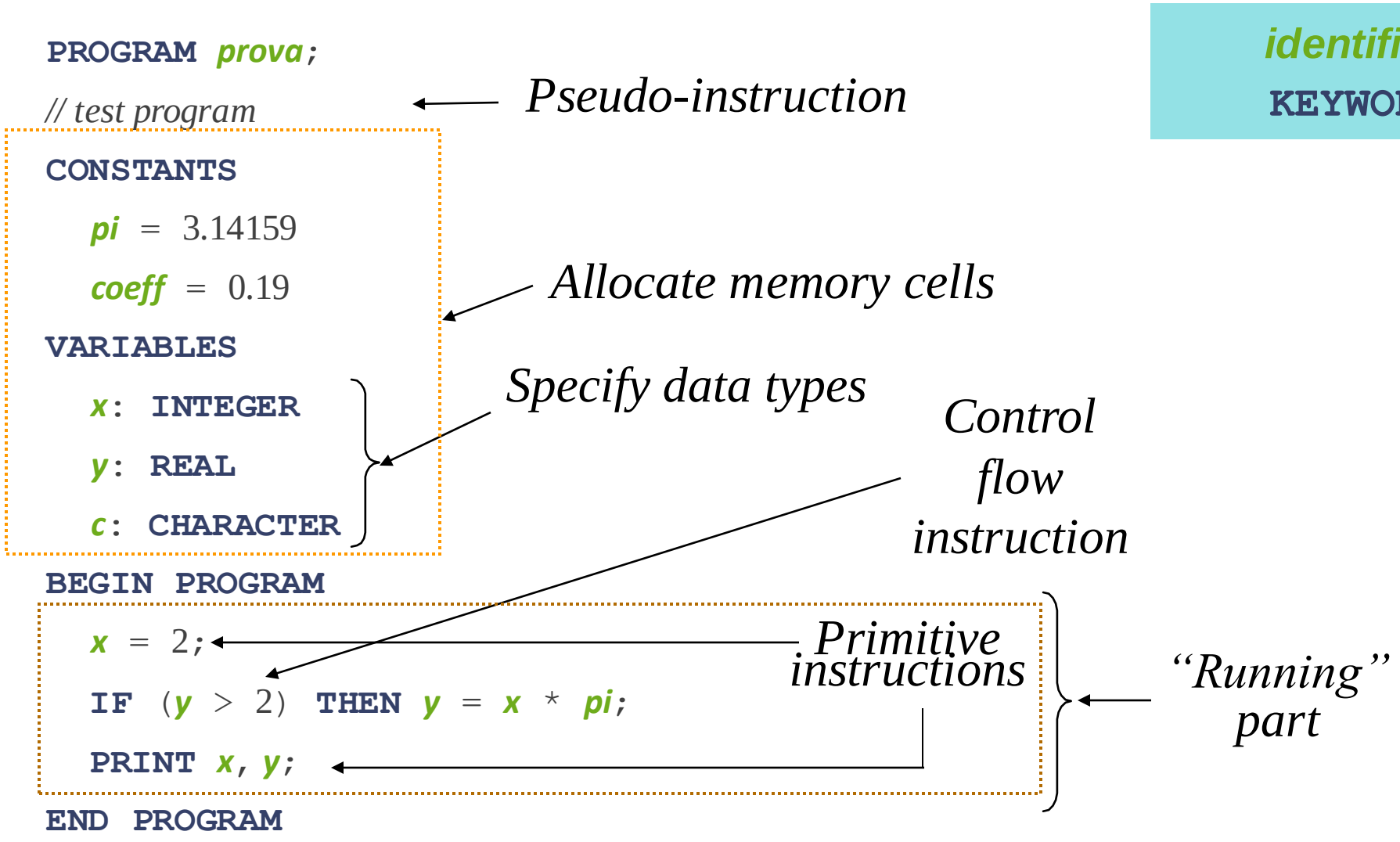
- Data are stored (as bits) in **memory**, at certain addresses
  - We'll see this more in detail later
- They usuallt have a **type**, and different types take different amounts of memory



Instructions

- Operations supported by the programming language, that will execute them by translating them to the machine level
  - Pseudo-instructions
    - Directive to the language interpreter or compiler, do not correspond to executable code
  - Basic (elementary, primitive) instructions
    - Access and modify data
    - Interact with I/O devices
    - etc.
  - Flow control instructions
    - Control the execution sequence of basic instructions

Program example



Names of “entities” in a program

- Let’s distinguish between
  - **Identifiers**
    - Name that we assign to **the data** that we use in a program
    - Ex: `x, y`, etc
  - **Keywords**
    - Words with a **special meaning** in a certain programming language
    - They are reserved and cannot be used for anything else
    - Ex: “`print`” in many languages, indicates an **output operation**

Introducing Python

1.3, 1.4, 1.5, 1.6

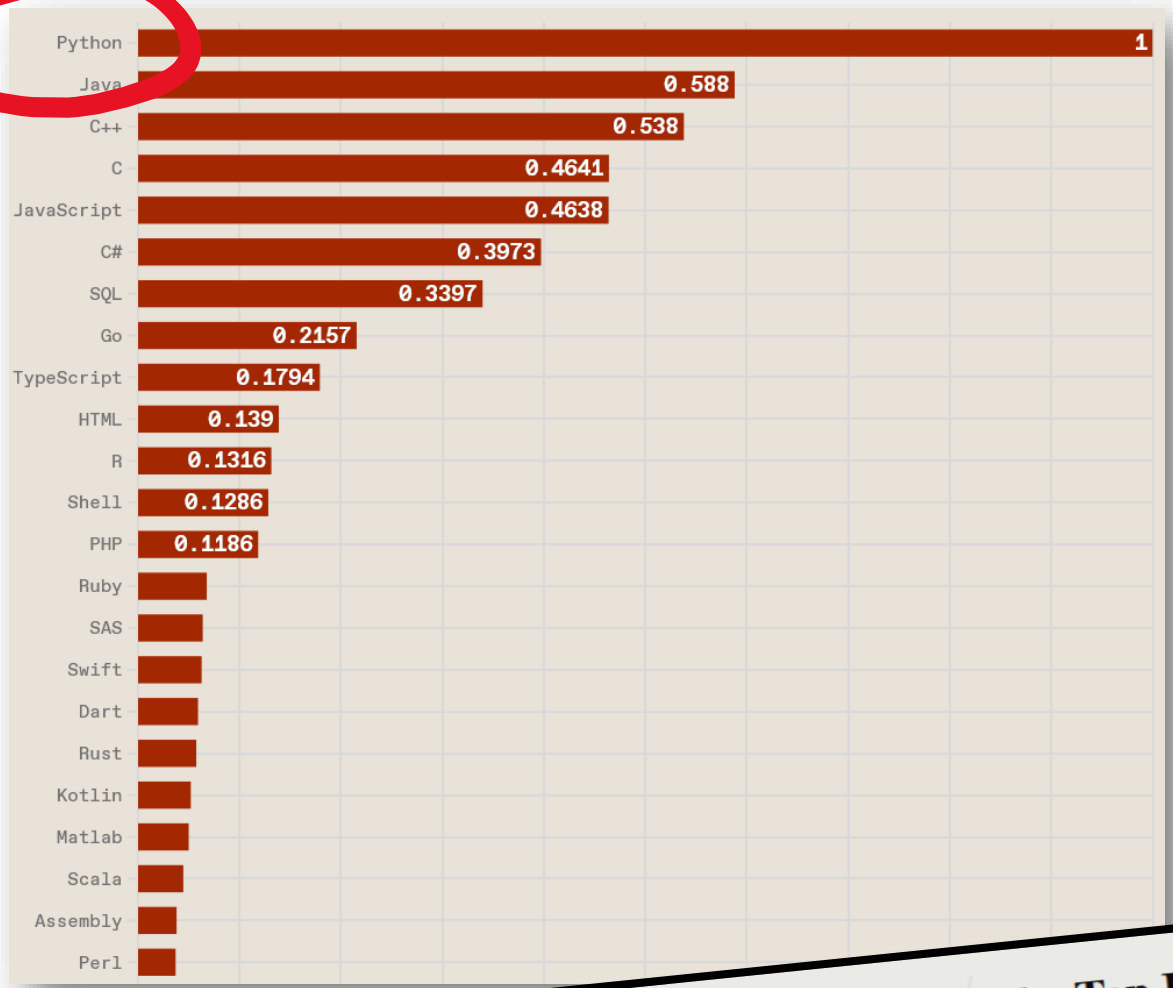
The Python Language

- Designed in the early 1990's by **Guido van Rossum**

- Van Rossum was dissatisfied with the languages available
  - They were optimized to write large programs that executed quickly

- He needed a language could be used to **create programs quickly** and also make them **easy to modify**

Why Python?



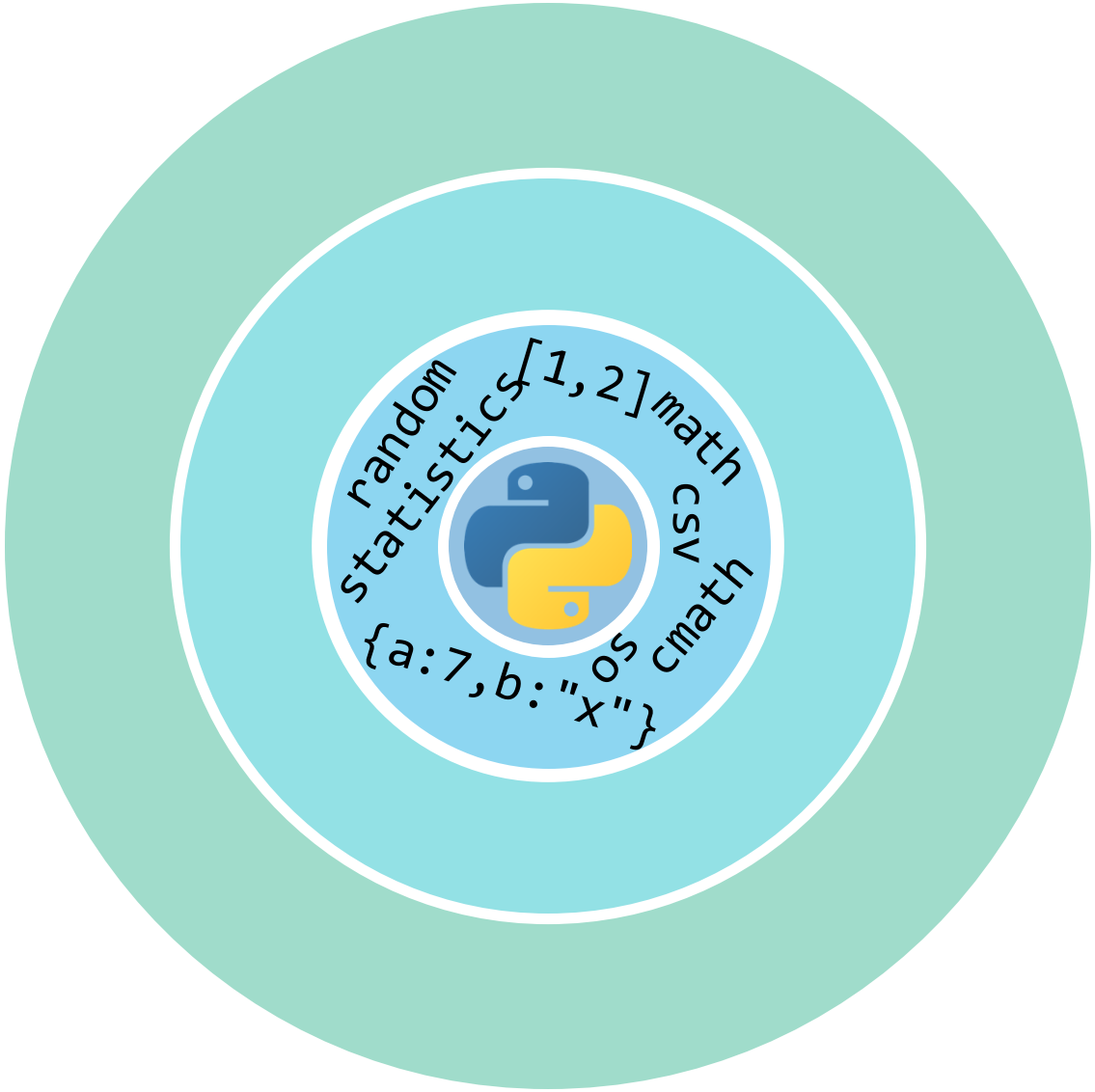
IEEE SPECTRUM

IEEE Spectrum | The Top Programming Languages 2023

Why Python?

- Free and open source language
  Available for all operating systems
  - Windows, Mac OS X, Linux
  - Embedded Systems, Raspberry PI, Android
- Simple, clean , regular syntax
- Battery included approach
  - Extensive library of standard functions
- Low entry step
- Endless online documentation

Batteries included

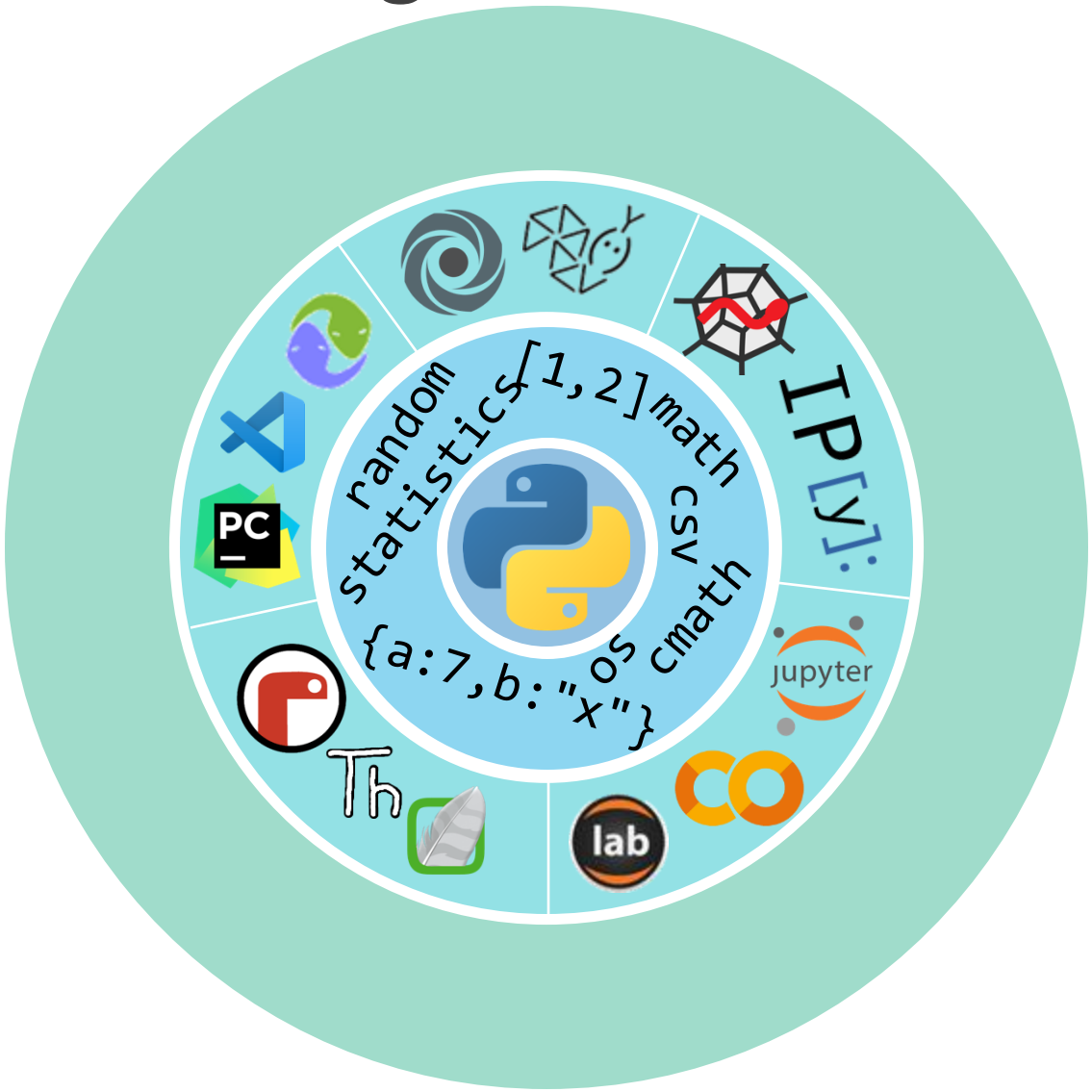


- Standard data types
  - boolean , int, float, complex, string, regexp , …
- Advanced containers
  - lists, tuples, sets, dictionaries, …
- Object-oriented
- 200+ modules in the standard library

Standard library

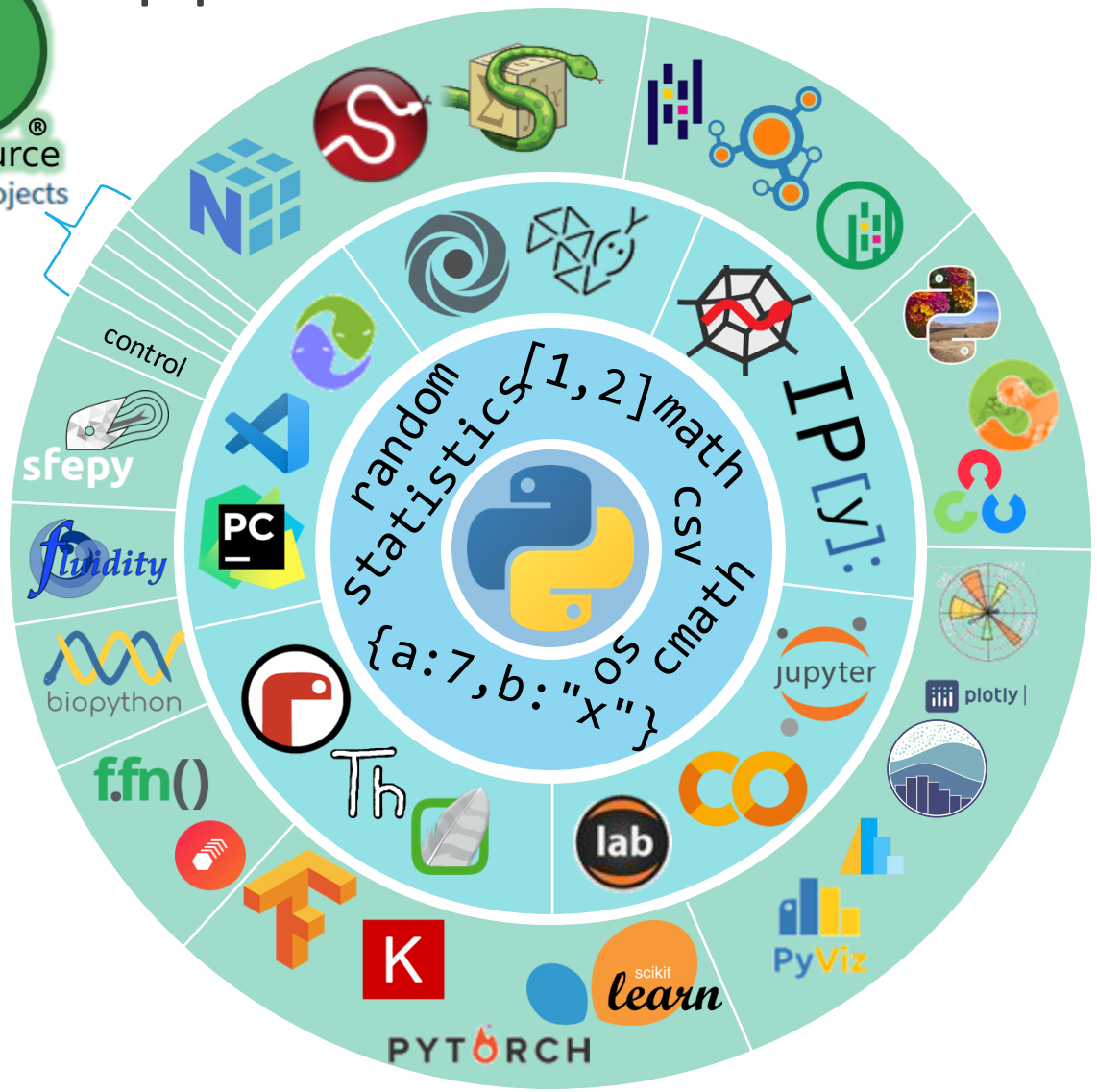
abc	chunk	**decimal**	getpass	keywords	optparse	**queue**	sndhdr	telnetlib	**unittest**
aifc	**cmath**	difficult	gettext	line-cache	**os**	quopri	socket	**tempfile**	urllib
argparse	cmd	dis	globe	local	ossaudiodev (Linux, FreeBSD)	**random**	socketserver	termios (Unix)	uu
array	**codecs**	distutils	**graphlib**	**logging**	parser	**king**	spwd (Unix)	**test**	**uuid**
ast	codeop	doctest	grp (Unix)	lzma	**pathlib**	readline (Unix)	**sqlite3**	**textwrap**	venv
assync	**collections**	**e-mail**	gzip	mailbox	pdb	reprlib	ssl	threading	warnings
asyncio	colorsys	encodings	**hashlib**	email cap	pickle	resources (Unix)	stat	**time**	wave
asyncore	compileall	ensurepip	**heapq**	marshal	pickletools	rlcompleter	**statistics**	**timeit**	weakref
atexit	configparser	list	hmac	**math**	pipes (Unix)	runpy	**string**	**tkinter**	**web browser**
audio op	contextlib	errno	**html**	mimetypes	pkgutil	card	stringprep	token	winreg (Win)
base64	contextvars	faulthandler	**http**	mmap	platform	secrets	struct	tokenize	winsound (Win)
bdb	**copy**	fcntl (Unix)	imaplib	modulefinder	plistlib	select	subprocess	trace	wsgiref
binascii	copyreg	**filecmp**	imghdr	msilib (Windows)	poplib	selectors	sun	traceback	xdrlib
binhex	crypt (Unix)	fileinput	imp	msvcrt (Windows)	**pprint**	shelve	symbol	tracemalloc	**xml**
bisect	**csv**	fnmatch	importlib	multi-processing	profile	shlex	symtable	tty (Unix)	xmlrpc
built-ins	ctypes	**fractions**	inspect	netrc	pstats	**Shut up**	**sys**	**turtle**	**zipapp**
bz2	curses (Unix)	ftplib	**I**	nis (Unix)	pty (Linux)	signal	sysconfig	turtledemo	zip file
calendar	**data classes**	functools	ipaddress	nntplib	password (Unix)	site	syslog (Unix)	types	zipimport
cgi	**datetime**	gc	**itertools**	**numbers**	pyclbr	smtpd	tab-nanny	typing	**zlib**
cgitb	dbm	getopt	json	**operator**	pydoc	smtplib	Tarfile	unicodedata	zoneinfo

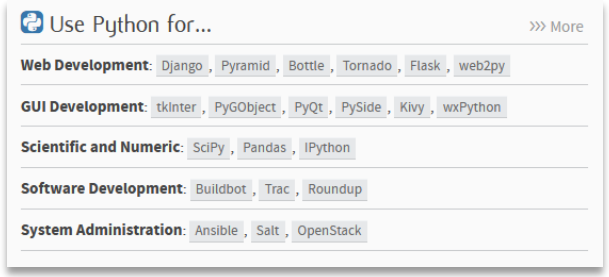
Working Environments



- Traditional IDEs
  - IDLE, PyCharm , Visual Studio Code, Eclipse PyDev , Spyder , …
- Interactive
  - IPython
- Notebook (also online)
  - Jupyter , JupyterLab , Google Colab
- Online resources
  - Repl.it, PythonAnywhere , Python Tutor
- Learning environments
  - Mu, Thonny , Wing

Application libraries





- Scientific computation
  - NumPy , SciPy , SymPy
- Data Analysis, Algorithms , Graphs
  - Pandas , networkx , GeoPandas
- Image Processing
  - Pillow , scikit -image, OpenCV
- Visualization
  - Pyviz , matplotlib , plotly , seaborn , altair
- Machine Learning
  - Scikit-learn , tensorflow , pytorch , keras
- Fintech
  - f.fn , zipline , pyalgotrade
- Biology and Genome
  - Biopython
- Fluid Dynamics
  - Fluidity
- Finished Elements
  - Sphepy
- Control systems

python™ Package Index

Individual Modules

ANACONDA®

Complete Toolkit for Data Science

Example: Scientific Libraries

- NumPy

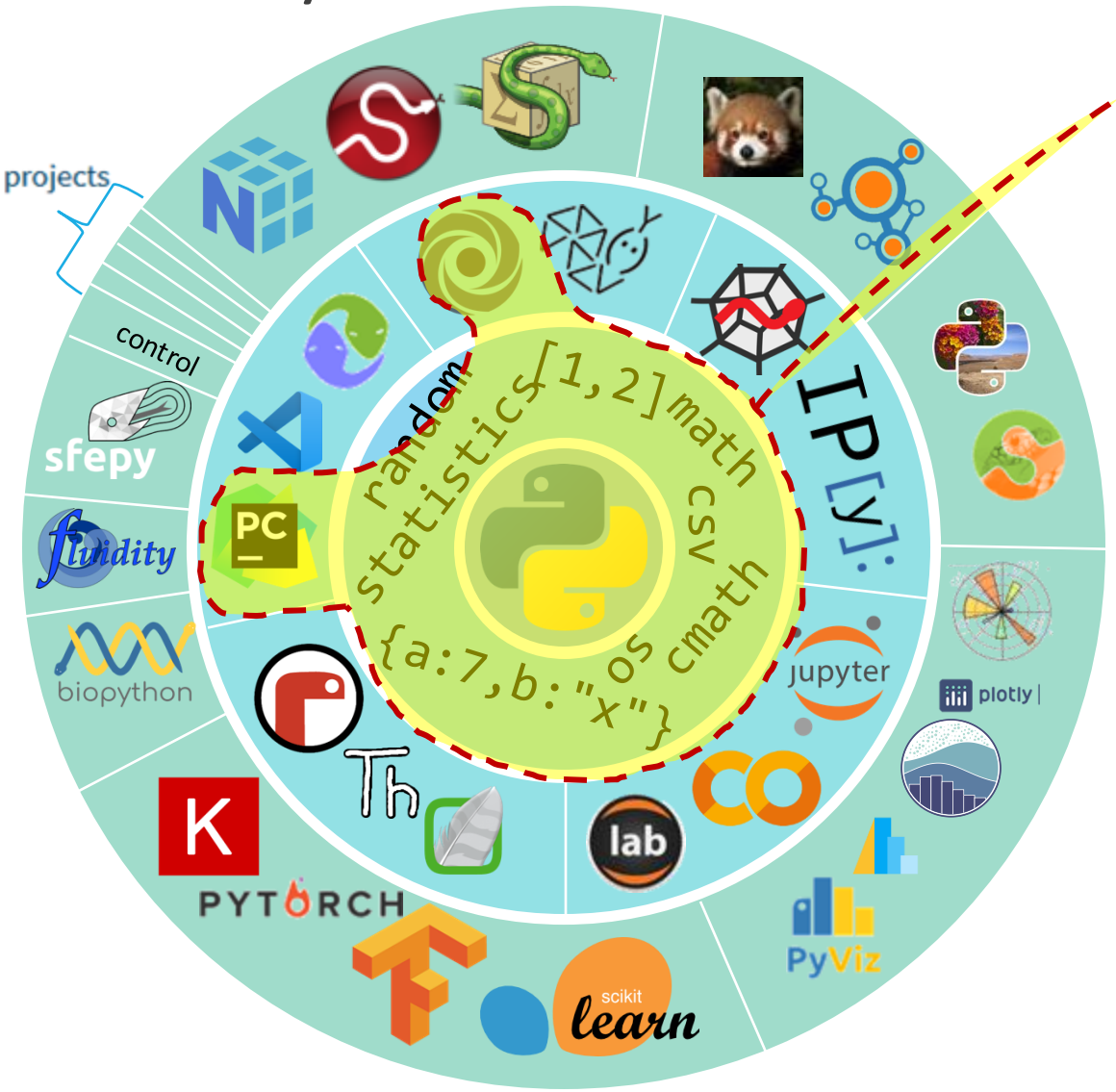
- SciPy

- SymPy

- Pandas

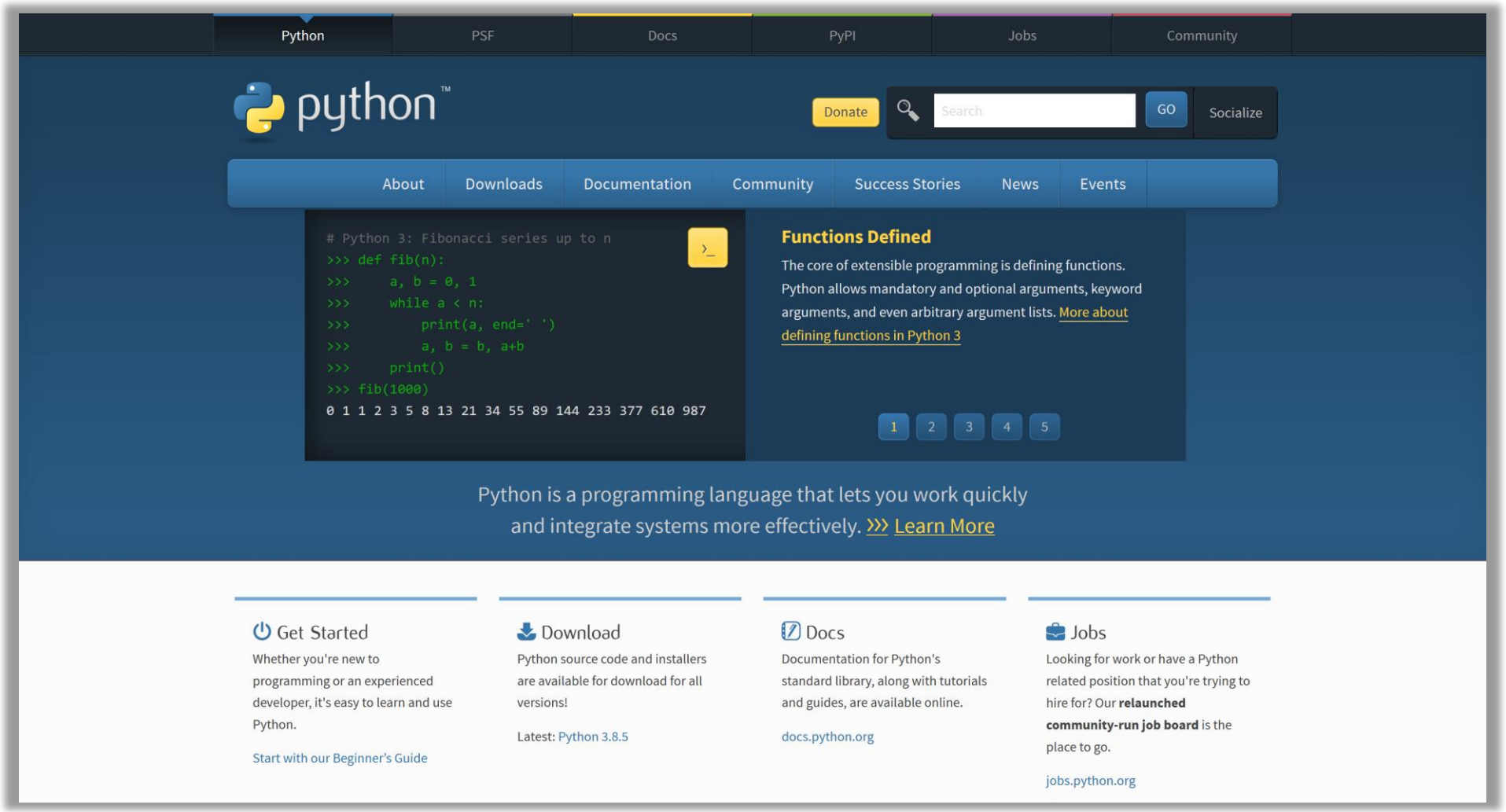
Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

First year

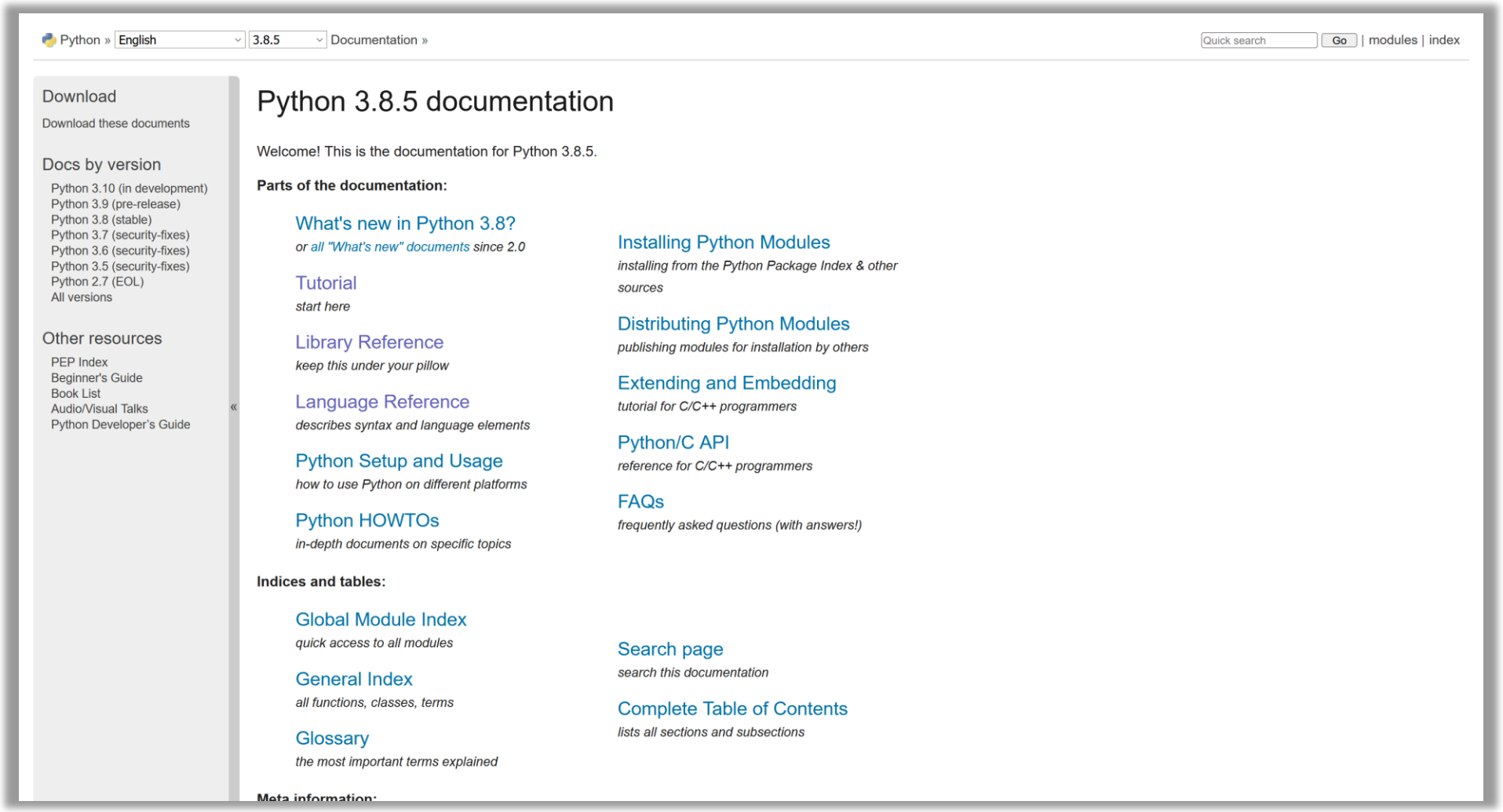


- Computer Science

https://www.python.org/



https://docs.python.org/



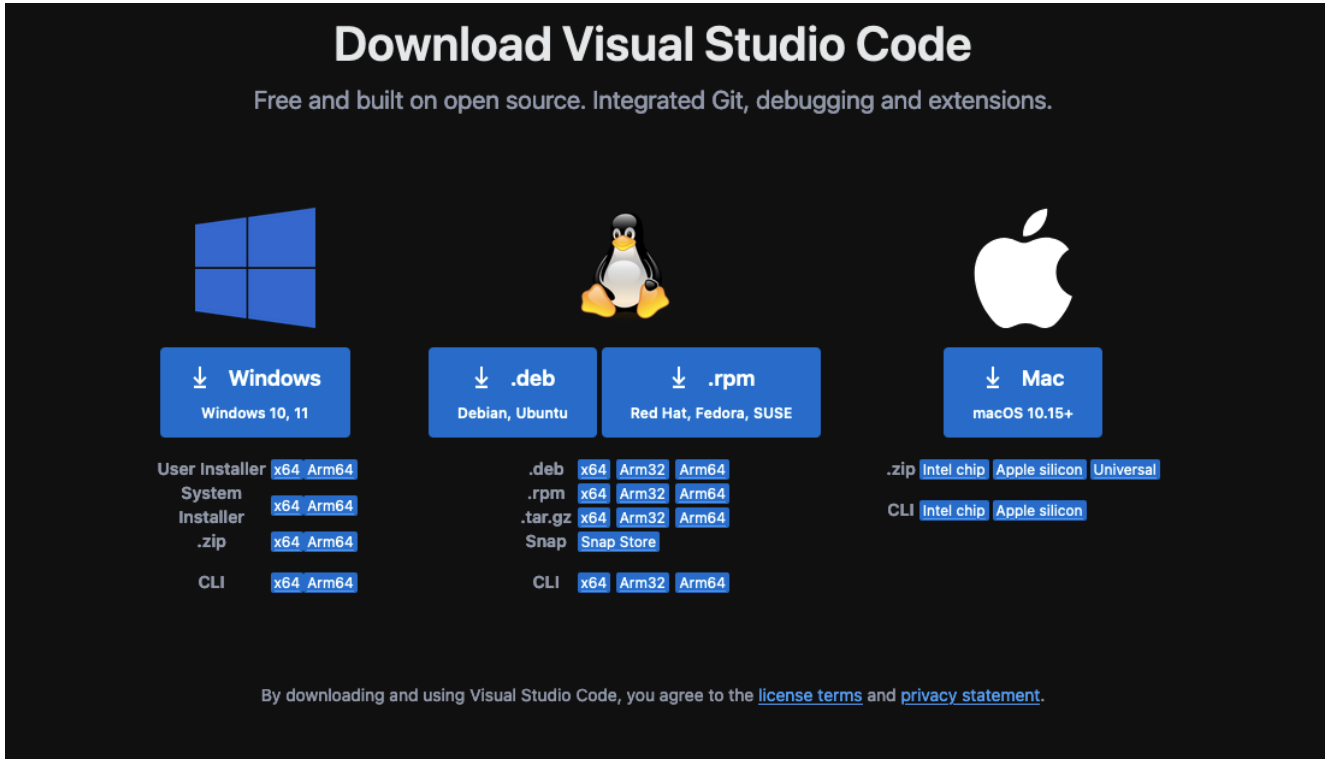
Programming Environments

- There are several ways of creating a computer program
  - Using an **Integrated Development Environment (IDE)**
    - IDLE, PyCharm, Visual Studio Code, …
  - Using a text editor
    - Notepad, Notepad++, Atom, vi, gedit, …

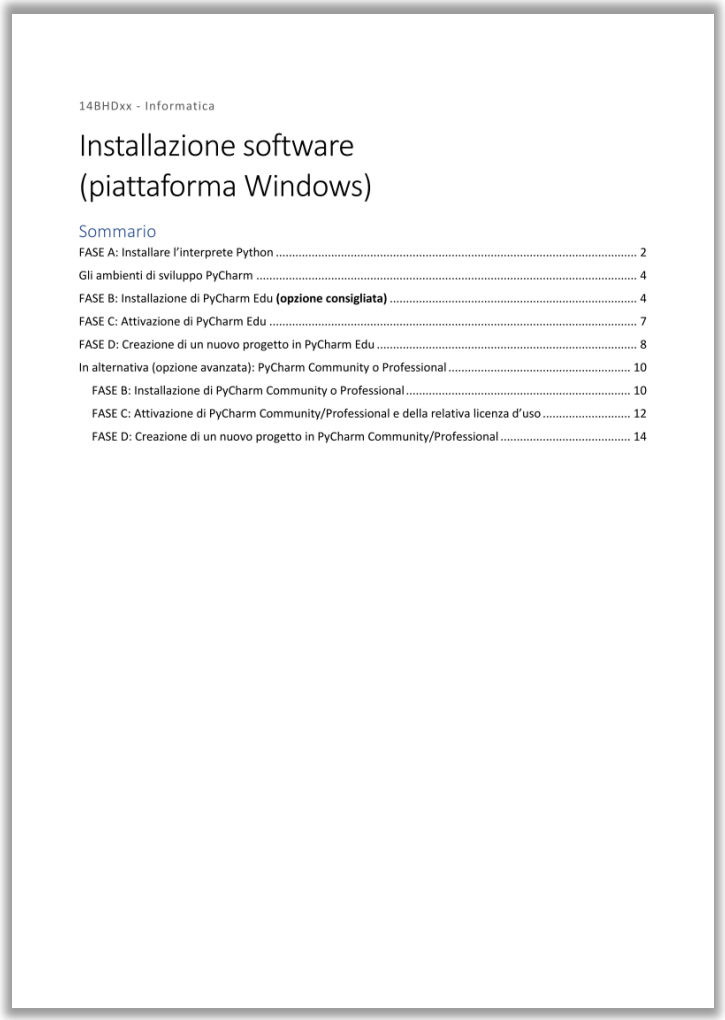
- In this course we'll use mainly the **Visual Studio Code IDE**.
  - Abbreviated VSCode

IDE

- Install Visual Studio Code and the Python Extension
- **https://code.visualstudio.com/download**



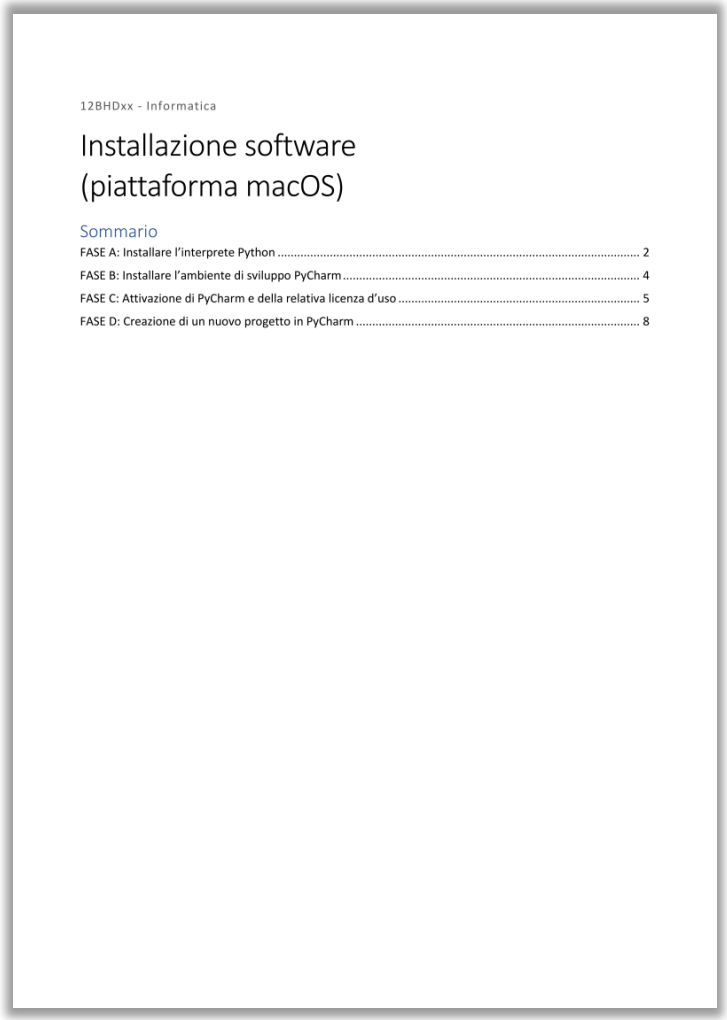
Installation instructions (in the Course Page)



14BHDxx - Informatica

Installazione software
(piattaforma Windows)

Sommario



12BHDxx - Informatica

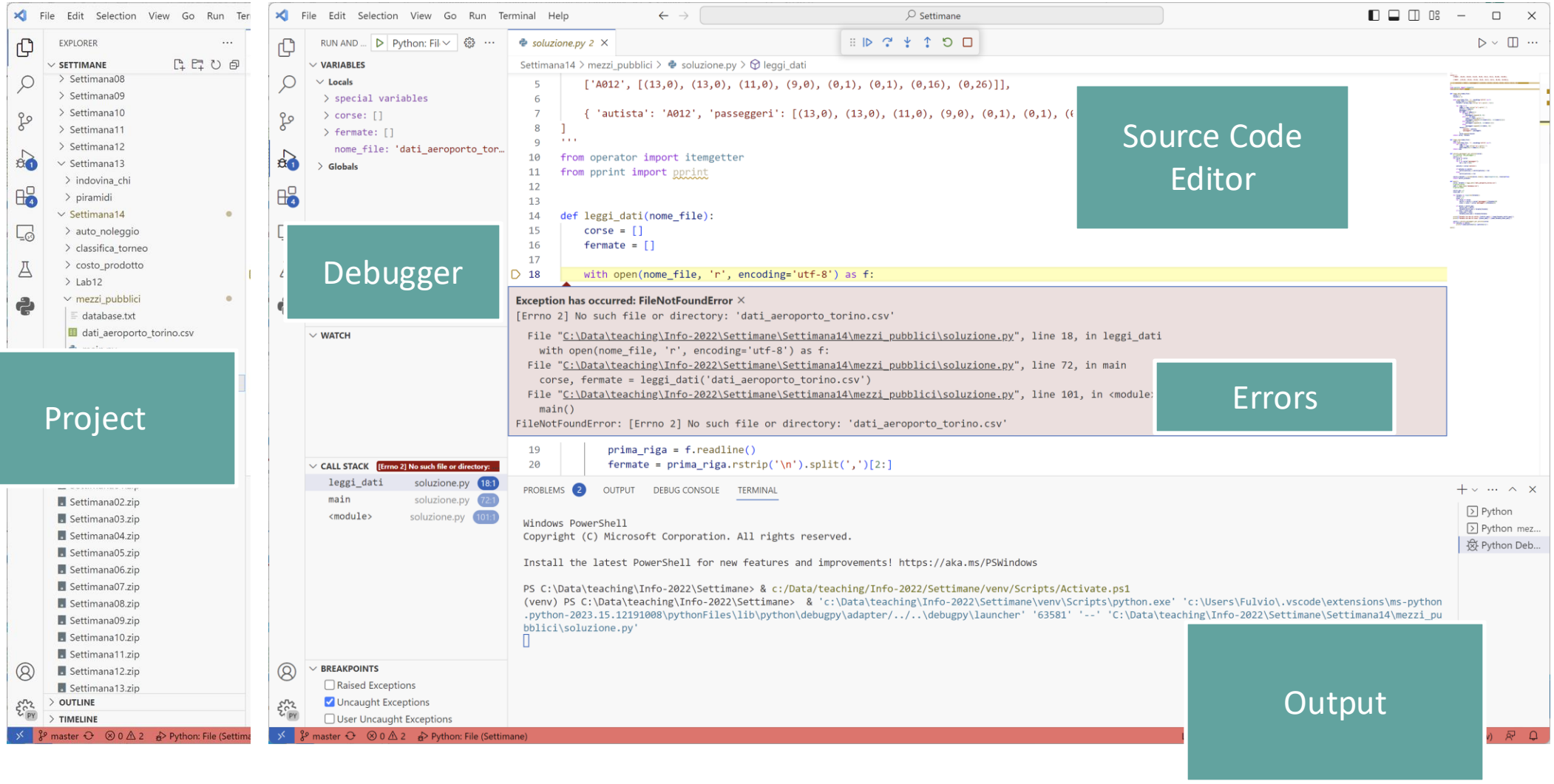
Installazione software
(piattaforma macOS)

Sommario

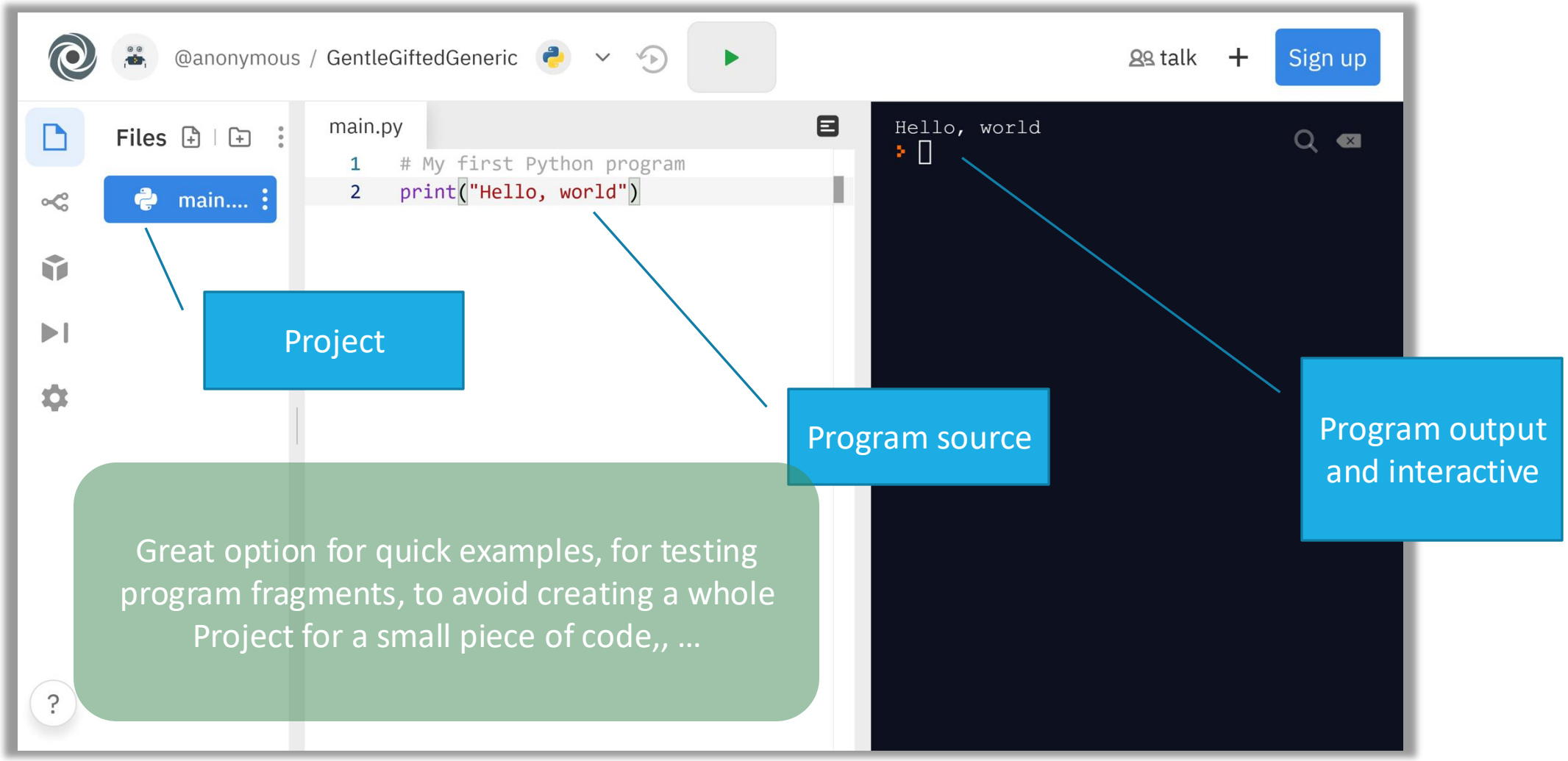
IDE components

- Most IDEs (not only VSCode) usually include:
- **The source code editor**: a text editor with additional functionality
  - Listing line numbers of code
  - Syntax highlighting and coloring (comments, text…)
  - Auto-indent source code
  - Highlight syntax errors
  - Auto-completion
- **Output window**
  - The output generated by the program
- **Debugger**
  - Tools to help you search and correct logic errors in the program

The Visual Studio Code IDE

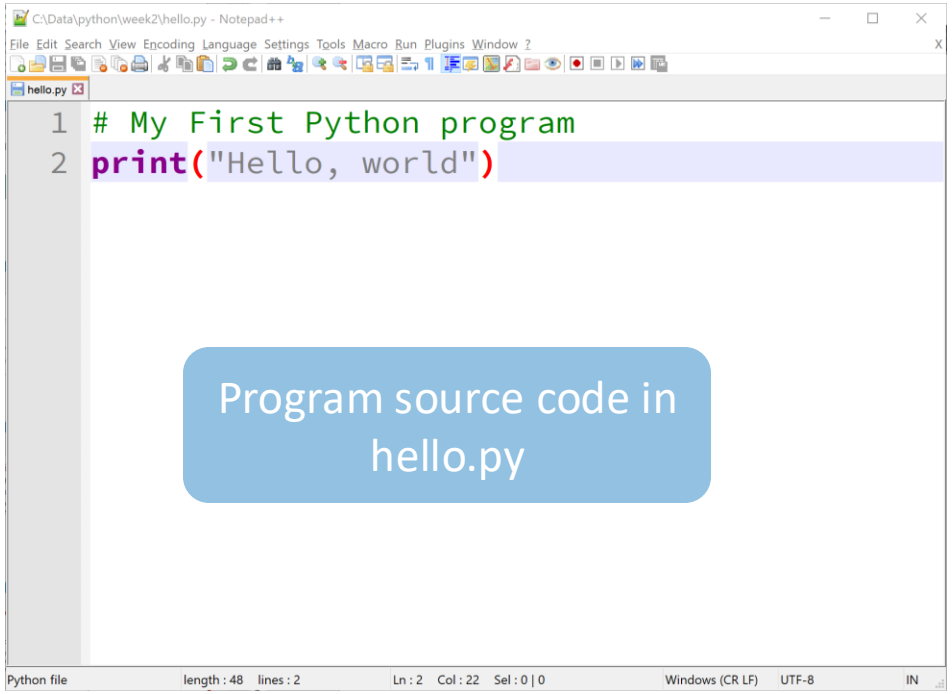


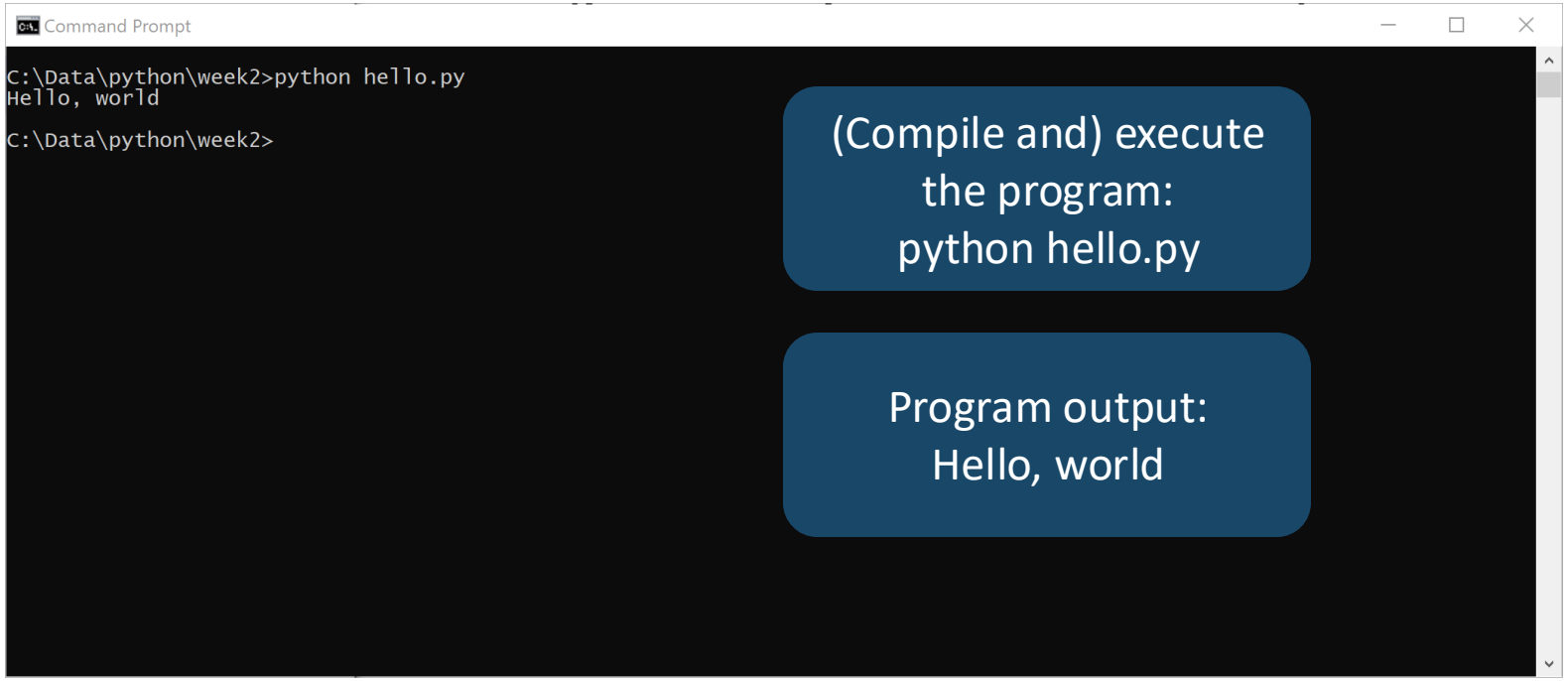
IDE Online: https://replit.com/



Text editor programming (not recommended)

- You can use a simple text editor to write your source code
- Once saved as hello.py, you can use a console window to:
  - Compile & Run the program

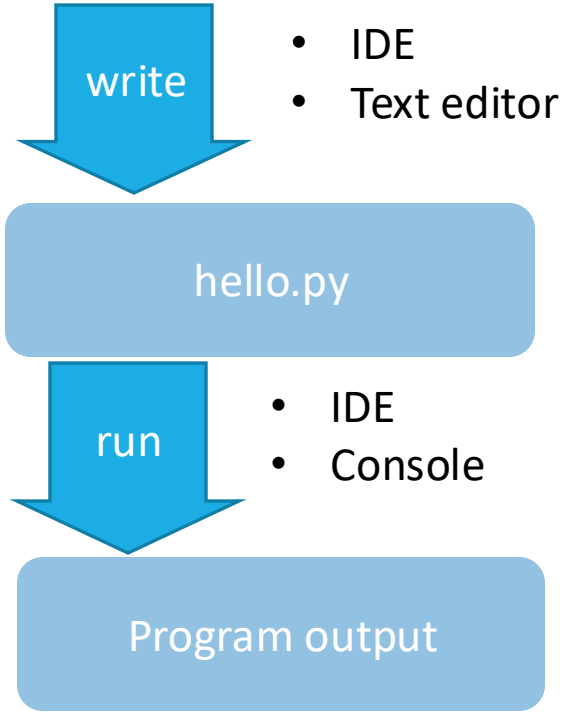




Your first program

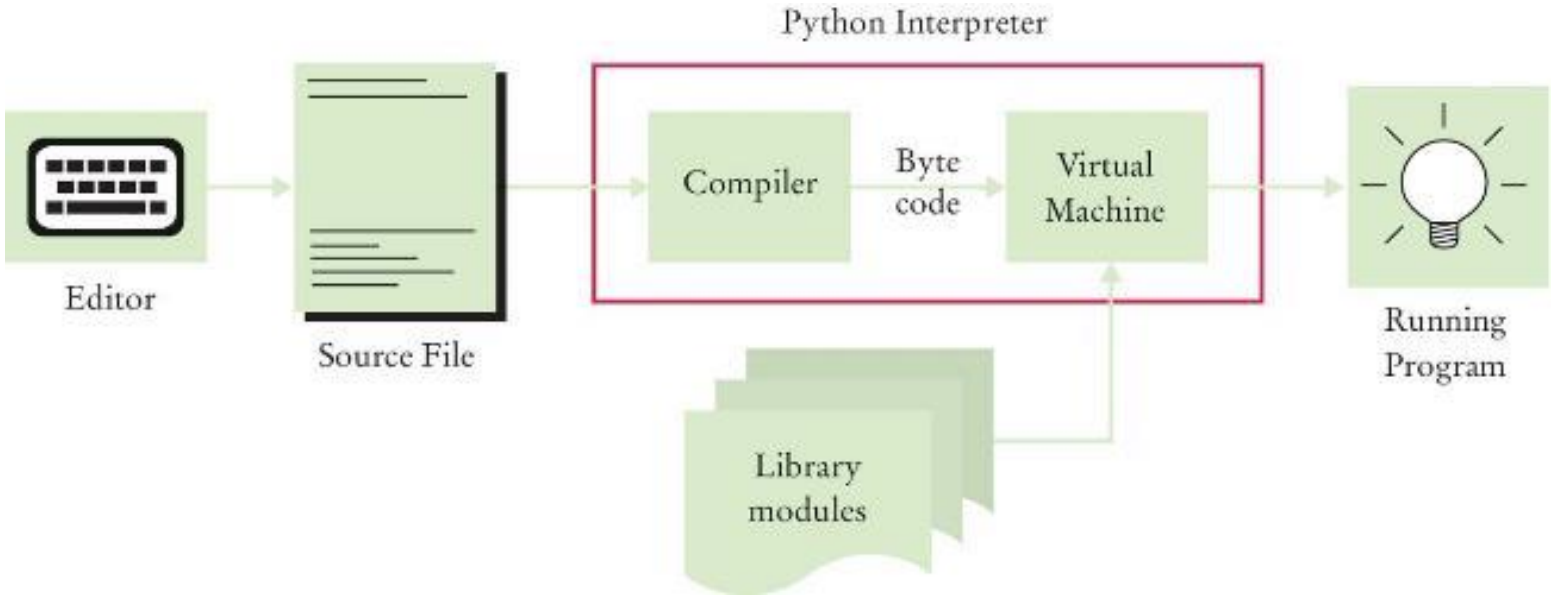
- Traditional ‘Hello World’ program in Python
  - `print` is an example of a Python statement

```
1  #My first program
2  print("Hola Mundo!!")
3
```



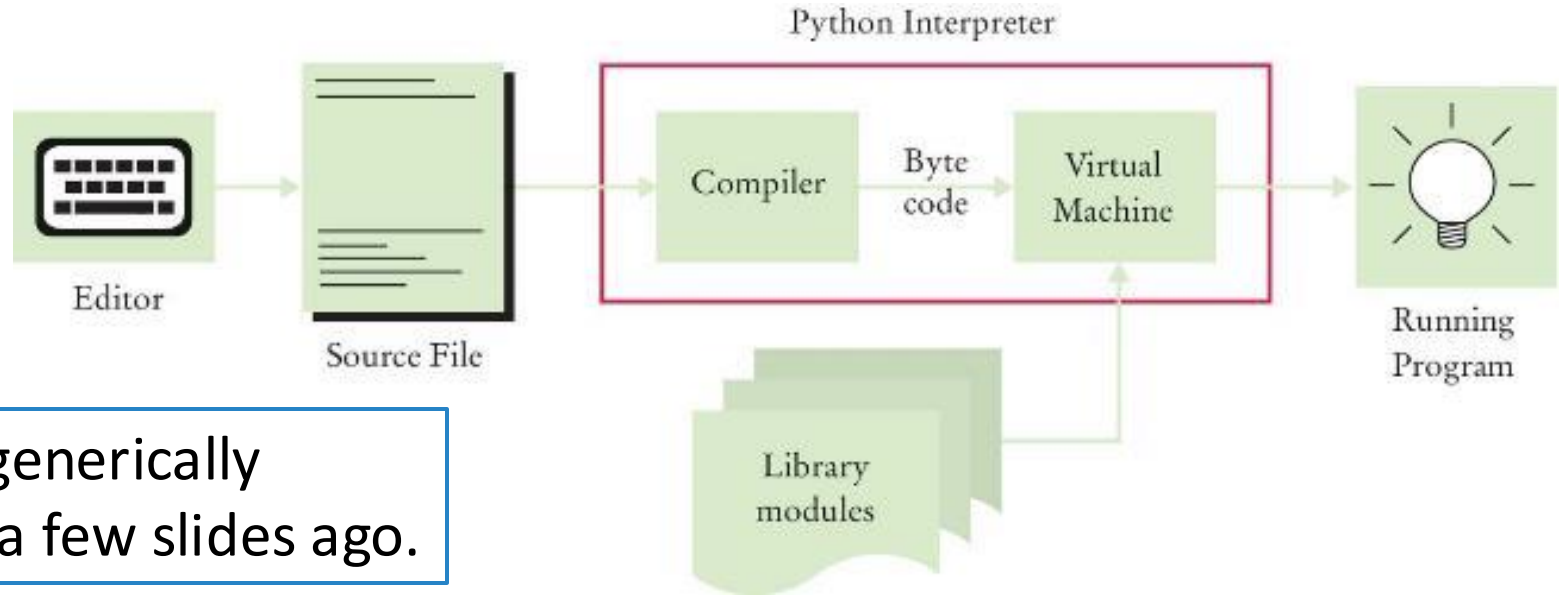
Source Code to a Running Program

- The **compiler** reads your program and generates **byte code instructions** (simple instructions for the Python Virtual machine)
  - The Python Virtual machine is a program that is similar to the CPU of your computer
  - Any necessary libraries (e.g., for drawing graphics) are automatically located and included by the virtual machine



Source Code to a Running Program

- The **compiler** reads your program and generates **byte code instructions** (simple instructions for the Python Virtual machine)
  - The Python Virtual machine is a program that is similar to the CPU of your computer
  - Any necessary libraries (e.g., for drawing graphics) are automatically located and included by the virtual machine



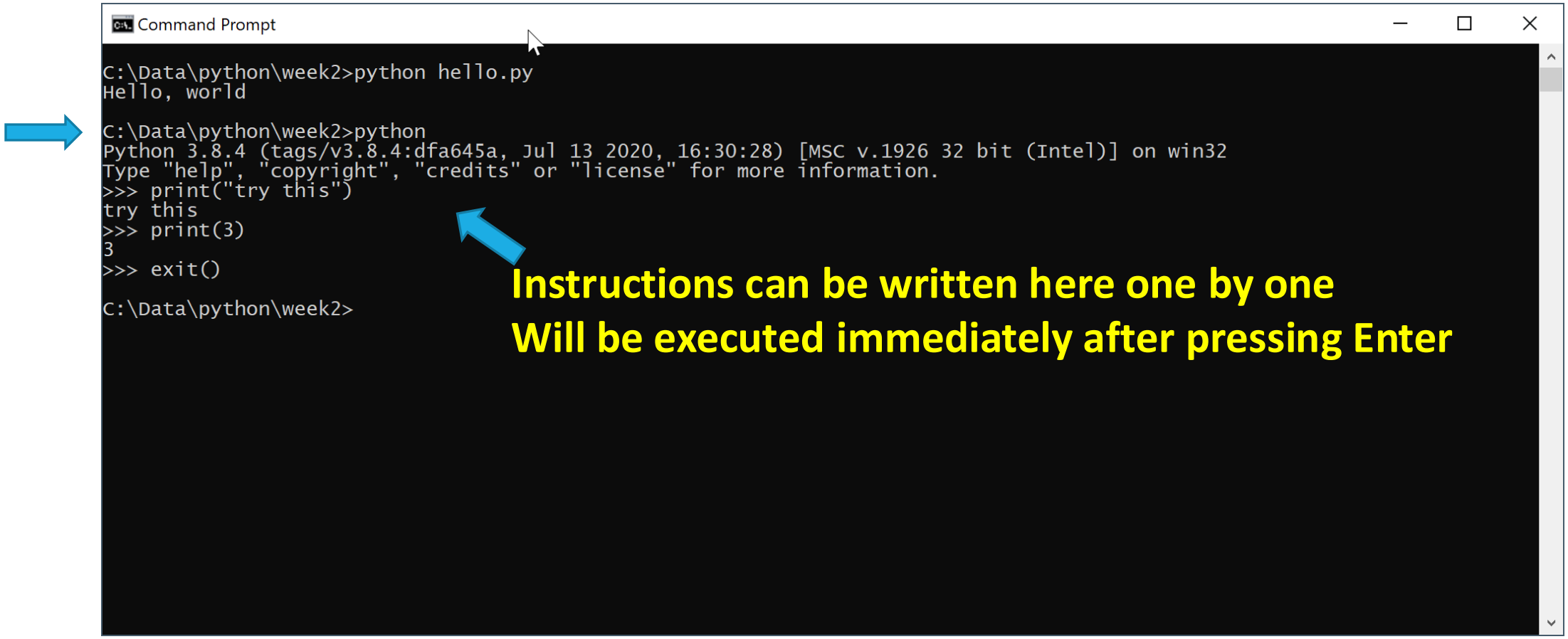
We called it generically "Translator" a few slides ago.

Python Interactive Mode

- The Python interpreter can load a file and execute all the instructions contained in it (and other files from the same project)

- Alternatively: in **interactive mode**, you can run instructions one at a time
  - It allows quick ‘test programs’ to be written
  - Try and experiment instructions

Python Interactive Mode

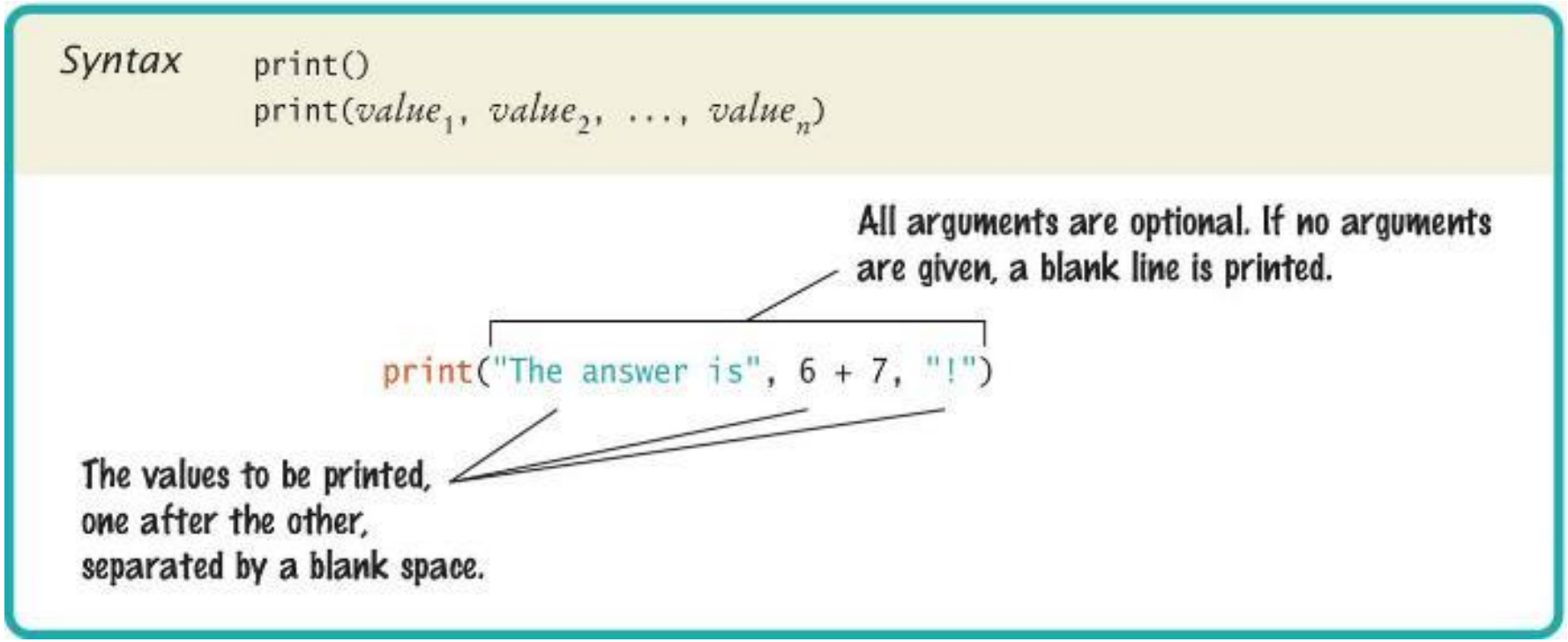


Writing/typing a Python program

- Careful about spelling - e.g., ‘`print`’ vs. ‘`primt`’
- PyTHon iS CaSe SeNsItiVe
- Spaces are important, especially at the beginning of a line (indentation)
- Lines beginning with # are comments (ignored by the interpreter)

Basic Python Syntax: Print

- Using the Python `print()` function
  - A function is a collection of programming instructions (with a **name**) that carry out a particular task (in this case to print a value onscreen)
  - It's code that somebody else wrote for you!



Syntax for Python Functions

- To use, or **call**, a function in Python you need to specify:
  - The name of the function that you want to use
    - In the previous example, the name was `print`
  - All values (arguments) needed by the function to carry out its task
    - in this case, `"Hello World!"`
  - Arguments are enclosed in parentheses
  - Multiple arguments are separated with commas.

  - If you want to print some **text**, you have to enclose it between **'single quotes'** or **"double quotes"**
    - This will create a **string** → We'll see later what it means.

More examples of the `print` function

- Printing numerical values
  - `print(3 + 4)`
  - Evaluates the expression 3 + 4 and displays 7
- Passing multiple values to the function
  - `print("The answer is", 6 * 7)`
  - Displays `The answer is 42`
  - Each value passed to the function is displayed, one after another, with a blank space after each value
- By default the print function starts a new line after its arguments are printed
  - `print("Hello")`
  - `print("World!")`
  - Prints two lines of text:
    - `Hello`
    - `World!`

Our Second Program (printtest.py)

```
# Computes and Prints 7
print(3 + 4)

# Prints "Hello World!" in two lines
print("Hello")
print("World!")

# Prints multiple values with a single print function call
print("My favorite numbers are", 3 + 4, "and", 3 + 10)

# Prints three lines of text with a blank line
print("Goodbye")
print()
print("Hope to see you again")
```

Errors

- When programming, we may insert **errors** in the source code
- More often than you think…
- Most of a programmer's time is spent finding and correcting errors (debugging)

Errors

COMPILE-TIME ERRORS

OR **SYNTAX** ERRORS

- Spelling, capitalization, punctuation
- Ordering of statements, matching of parenthesis, quotes, indentation, …
- **No executable program is created by the compiler**
- **Correct first error listed, then compile again**
  - Repeat until all errors are fixed
- **Usually detected and highlighted by the IDE**

RUN-TIME ERRORS

OR **LOGIC** ERRORS

- **The program runs, but produces unintended results**
- **The program may ‘crash’**
- **The most difficult to find and correct**
  - Even for experienced programmers

Syntax Errors

- Syntax error are caught by the compiler
- What happens if you
  - Miss-capitalize a word `Print("Hello World!")`
  - Leave out quotes `print(Hello World!)`
  - Mismatch quotes `print("Hello World!')`
  - Don’t match brackets `print('Hello'`
- Type each example above in the IDE
  - In the program source code
  - In the interactive Python console
  - What error messages are generated?

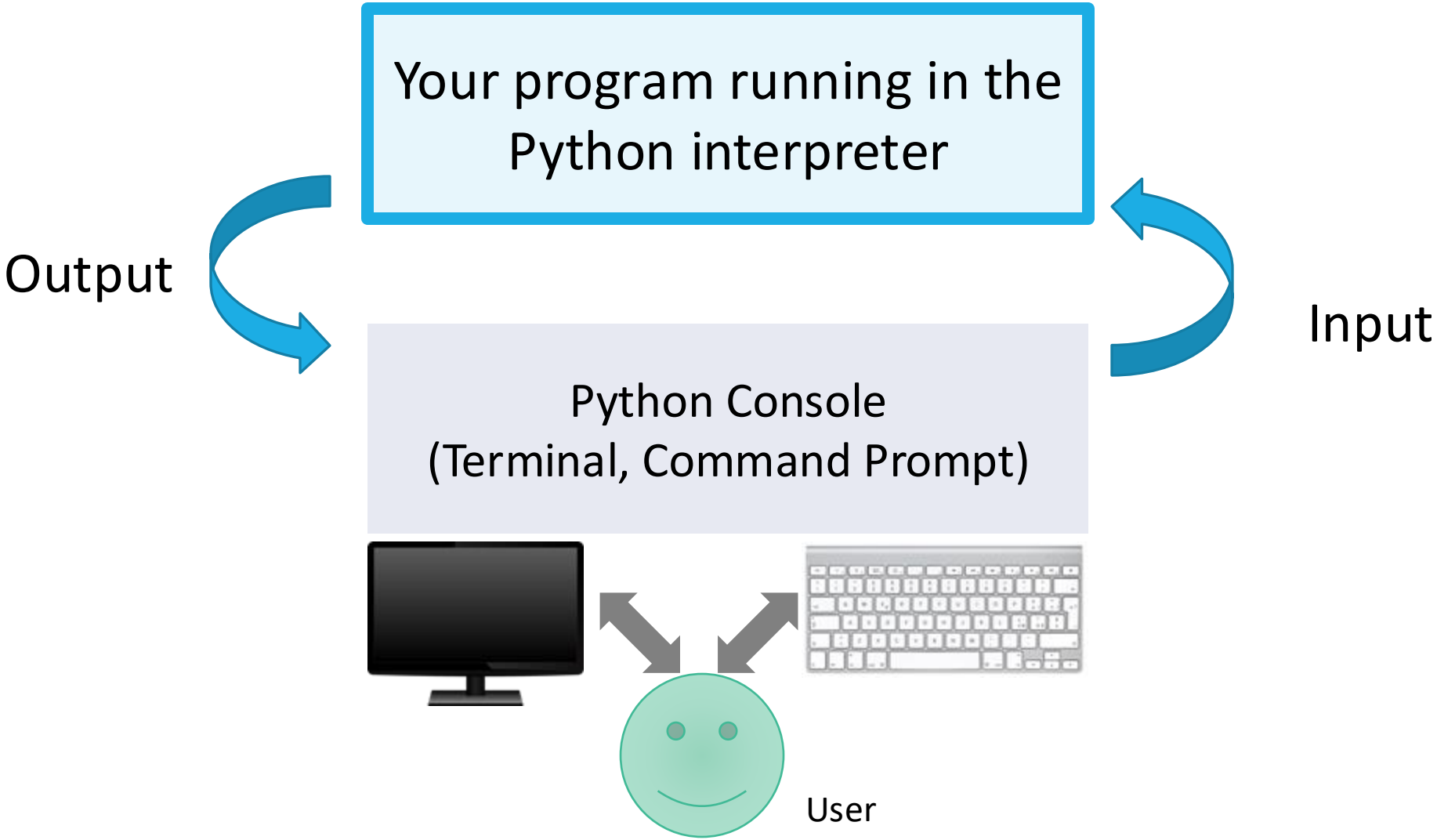
Logic Errors

- What happens if you
  - Divide by zero `print(1/0)`
  - Misspell output `print("Hello, Word!")`
  - Forget to output Remove line 2
- Programs will compile and run
  - The output may not be as expected
- Type each example above in the IDE
  - What error messages are generated?

Input (Preview)

2.5

Input and Output



Input e Output

- We can read a **string** from the console using the `input()` function:
  - `name = input("Insert your name")`
- If you need a **numeric** input (instead of a string), you must **convert** the string:

```
ageString = input("Insert your age: ") # String
age = int(ageString) # Conversion to int
```

- …or in a single instruction:

```
age = int(input("Insert your age: "))
price = float(input("Insert the price: "))
```

This is an anticipation so you can use this stuff in Lab 1. We'll come back to it.

Thanks

- Part of these slides are [edited versions] of those originally made by **Prof Giovanni Squillero** (Teacher of Course 1)

License

- These slides are distributed under the Creative Commons license “Attribution - Noncommercial - ShareAlike 2.5 (CC BY-NC-SA 2.5)”
- You are free:
  - to reproduce, distribute, communicate to the public, exhibit in public, represent, perform and recite this work
  - to modify this work
- Under the following conditions:
  - **Attribution** — You must attribute the work to the original authors, and you must do so in a way that does not suggest that they endorse you or your use of the work.
  - **Non-Commercial** — You may not use this work for commercial purposes.
  - **Share Alike** — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or equivalent license as this one.
- http://creativecommons.org/licenses/by-nc-sa/2.5/it/