Basics of Computing with Python

A.Y. 2020/2021
Prof. Alessandro Ortis

ortis@dmi.unict.it

Basics of Computing

Prof. Alessandro Ortis

Dipartimento di Matematica e Informatica

(Office 362 – DMI Second Floor)

www.dmi.unict.it/ortis

Monday 10-13 Aula Multimediale DEI

Tuesday 15-18 Aula Multimediale DEI

Overall schedule: 60 hours

SYLLABUS

Introduction to programming

Programming languages

    Programming languages: machine, assembly and high level.

    Translation problem: compilation and interpretation.

    Installation of the development environment for the *Python* language. First program: Editing, Running, Debugging.

Constructs of Python language

    Basic syntax, data types, predefined operators, I/O management.

    Numbers and mathematical functions.

    Flow control: constructs of selection and iterative.

    Functions.

Built-in data structures in Python

    Strings. Lists, Tuples, Dictionaries.

Advanced topics

    Notable algorithms: Searching, Sorting, Merging. Basics of computational complexity. Recursive functions.

    Modules. Basics on the mathematical libraries *NumPy* and *SciPy* and the graphic library *PlotPy*.

Syllabus:

http://syllabus.unict.it/insegnamento.php?id=16345

- Course goals:
    - To develop problem solving and programming skills to enable the student to design solutions to non-trivial problems and implement those solutions in Python.
    - To master the fundamental programming constructs of Python, including variables, expressions, functions, control structures, and arrays.
    - To build a foundation for more advanced programming techniques, including object-oriented design and the use of standard data structures

Books

A.Downey, *Think Python*, 2nd Ed., Grean Tea Press (online available).

M.Lutz, *Learning Python*, 4th Ed., O'Reilly
(online available).

D.Pine, *Introduction to Python for Science and Engineering*, SMTEBooks - CRC Press
online available).

Jessen Havill - Discovering Computer Science: Interdisciplinary Problems, Principles, and Python Programming Chapman and Hall/CRC;

Course Structure and Exams

All the material shown (slides and code fragments) is made available to students in the *Teams* repository of the course.

Video projector is used for lectures in the classroom. The slides are not intended to replace the reference texts but represent a precise guide to the course topics. Many lessons will be carried out in an interactive teacher-learners mode;

Practical classroom exercises are planned to familiarize the learners with the drafting of Python code;

Some case studies in the field of small software design will be discussed.

EXAMS: Programmming Scripts in Python

What did you actually learn?

- problem-solving: the ability to take a problem, break it into manageable pieces, design and organize a step-by-step solution
- programming: the ability to design and implement problem solutions in the form of programs that can be understood and executed by computers
- critical-thinking: the ability to analyze and identify the important features of a problem, systematically test and evaluate solutions
- communications: the ability to express ideas in a clear and precise manner, so that they could be understood by the computer (code) or another person (code & comments)

PROGRAMMING SKILLS

CRITICAL-THINKING   COMMUNICATION SKILLS

PROBLEM-SOLVING SKILLS
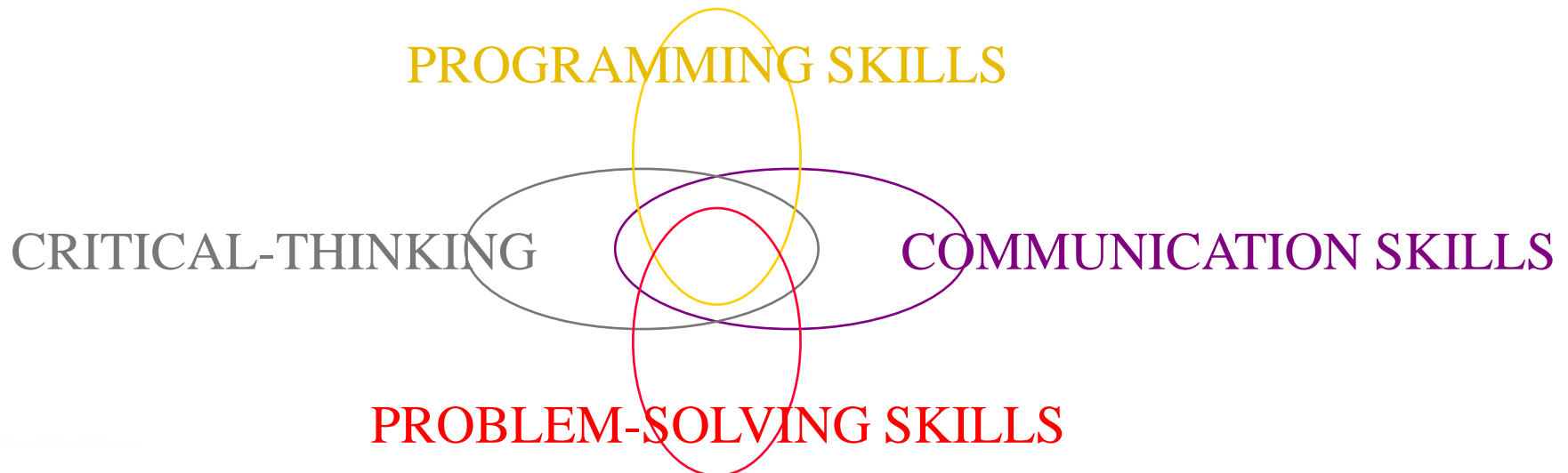
What is programming?

Programming is *applied problem-solving*

1. understand a problem
2. identify relevant characteristics
3. design an algorithm (step-by-step sequence of instructions to carry out a task)
4. implement the algorithm as a computer program
5. test the program by repeated (and carefully planned) executions
6. GO BACK AND REPEAT AS NECESSARY

in short: *programming* is the process of designing, writing, testing and debugging algorithms that can be carried out by a computer.

We encounter algorithms everyday: directions to dorm, instruction manual, recipe

- people are smart, so spoken languages can be vague
- computers are not smart, so programming languages are extremely picky

Problem-solving example

- Sudoku is a popular puzzle craze

  given a partially filled in 9x9 grid, place numbers in the grid so that
  - each row contains 1..9
  - each column contains 1..9
  - each 3x3 subsquare contains 1..9

how do people solve these puzzles?

if we wanted to write a program to solve Sudoku puzzles, must/should it use the same strategies?

Programming is a means to an end

- important point: programming is a tool for solving problems
  - computers allow people in many disciplines to solve problems they couldn't solve without them
    - — natural sciences, mathematics, medicine, business, …
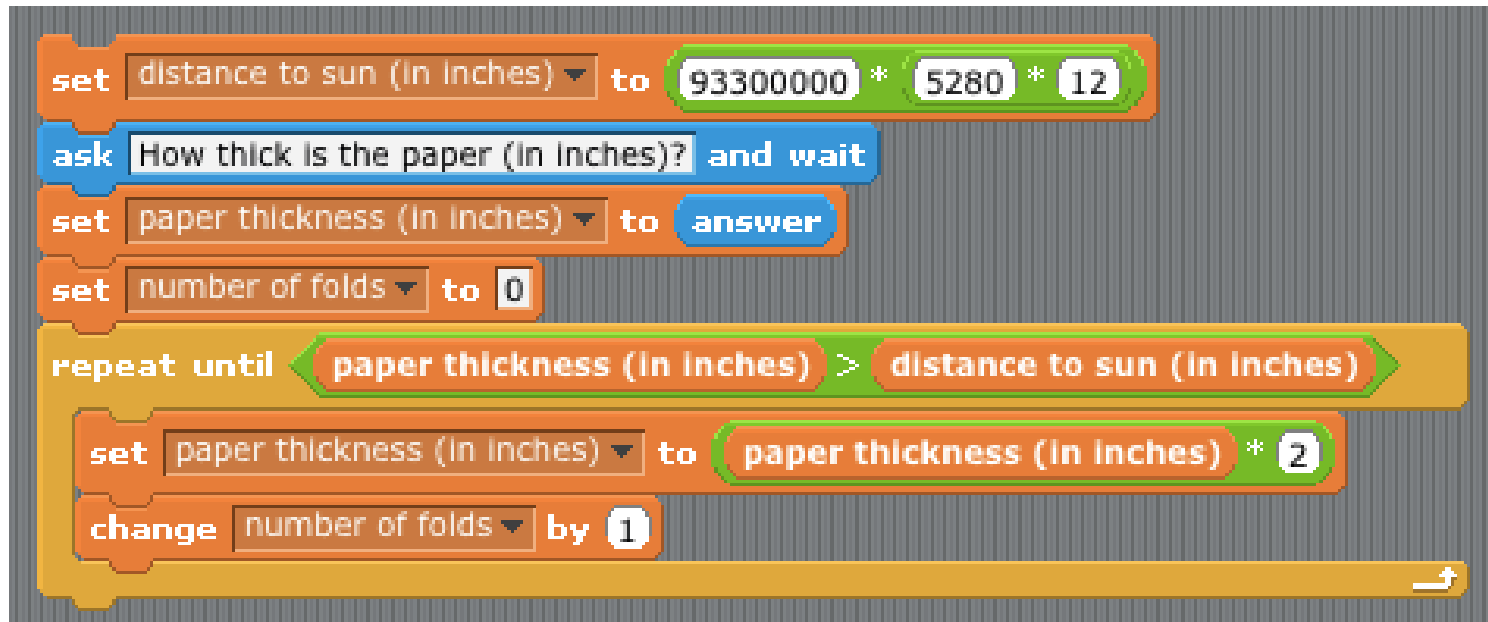 - to model this, many exercises will involve writing a program, then using it to collect data & analyze results

PAPER FOLDING PUZZLE: if you started with a regular sheet of paper and repeatedly fold it in half, how many folds would it take for the thickness of the paper to reach the sun?

- what information do you need (e.g., distance of sun)?
- what data values do you need to store and update?
- what is the basic algorithm?

recall, distance to sun is ~93.3 million miles

→ 93,300,000 mi x 5,280 ft/mi x 12 in/ft

→ 5,911,488,000,000 inches

```python
def paperFolding():
    """Solves the paper folding puzzle and displays the answer"""
    DISTANCE_TO_SUN = 93.3e6 * 5280 * 12

    currentThickness = float(input('How thick is the paper (in inches)? '))

    numFolds = 0
    while currentThickness < DISTANCE_TO_SUN:
        currentThickness *= 2
        numFolds += 1
    print(numFolds)
```

Ln: 13 Col: 0

Where do we start?

- explore programming concepts using graphical way to represent algorithms
  - we will explore your creative side, while building the foundation for programming

  - learn-by-doing, so be prepared to design & experiment & create

  - no previous programming experience is assumed

- will then segue into Python (v. 3) programming
  - transfer programming concepts into a powerful & flexible scripting language
  - classes will mix lecture and hands-on experimentation, so be prepared to do things!

hardware vs. software

basic terminology:

- hardware – the physical components of the computer
  - e.g.,   processor (Intel Core i5, AMD A6, Intel Pentium Mobile)
  - memory (RAM, cache, hard drive, floppy drive, flash stick)
  - input/output devices (keyboard, mouse, monitor, speaker)

- software – programs that run on the hardware
  - e.g.,   operating system (Windows 7, Mac OS X, Linux)
  - applications (Word, Excel, PowerPoint, RealPlayer, IE, Firefox)
  - development tools (JDK, BlueJ, .NET, IDLE, Scratch)

The easiest way to tell the difference between hardware and software is to kick it. If it hurts your toe, it's hardware.

Carl Farrell

History of computing technology

When were "modern" computers invented?
When were computers accessible/affordable to individuals?
When was the Internet born?
When was the Web invented?
How did Bill Gates get so rich?

the history of computers can be divided into generations, with
each generation defined by a technological breakthrough

0. gears and relays
  ➔ 1. vacuum tubes
    ➔ 2. transistors
      ➔ 3. integrated circuits
        ➔ 4. very large scale integration
   ➔ 5. parallel processing & networking

Generation 0: Mechanical Computers

- 1642 – Pascal built a mechanical calculating machine
    - used mechanical gears, a hand-crank, dials and knobs
    - other similar machines followed

1805 – the first programmable device was Jacquard's loom

- the loom wove tapestries with elaborate, programmable patterns
- a pattern was represented by metal punch-cards, fed into the loom
- using the loom, it became possible to mass-produce tapestries, and even reprogram it to produce different patterns simply by changing the cards
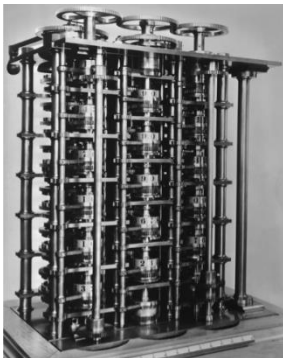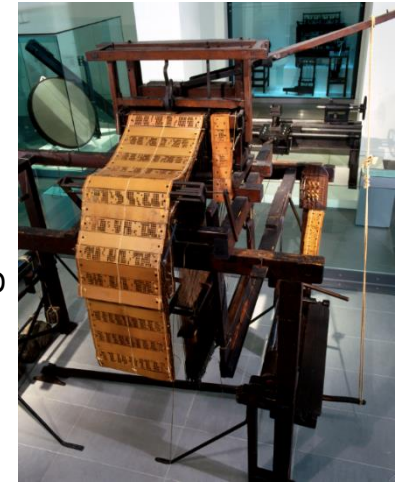
mid 1800's – Babbage designed his "analytical engine"

- its design expanded upon mechanical calculators, but was programmable via punch-cards (similar to Jacquard's loom)
- Babbage's vision described the general layout of modern computers
- he never completed a functional machine – his design was beyond the technology of the day

Generation 0 (cont.)

1930's – several engineers independently built "computers" using electromagnetic relays

- an electromagnetic relay is physical switch, which can be opened/closed via electrical current
- relays were used extensively in early telephone exchanges

- Zuse (Nazi Germany) – his machines were destroyed in WWII

- Atanasoff (Iowa State) – built a partially-working machine with his grad student

- Stibitz (Bell Labs) – built the MARK I computer that followed the designs of Babbage
  - limited capabilities by modern standards: could store only 72 numbers, required 1/10 sec to add, 6 sec to multiply
  - still, 100 times faster than previous technology

Generation 1: Vacuum Tubes

- mid 1940's – vacuum tubes replaced relays
  - a vacuum tube is a light bulb containing a partial vacuum to speed electron flow
  - vacuum tubes could control the flow of electricity faster than relays since they had no moving parts
 - invented by Lee de Forest in 1906

1940's – hybrid computers using vacuum tubes and relays were built

COLOSSUS (1943)
- first "electronic computer", built by the British govt. (based on designs by Alan Turing)
- used to decode Nazi communications during the war
- the computer was top-secret, so did not influence other researchers

ENIAC (1946)
- first publicly-acknowledged "electronic computer", built by Eckert & Mauchly (UPenn)
- contained 18,000 vacuum tubes and 1,500 relays
- weighed 30 tons, consumed 140 kwatts
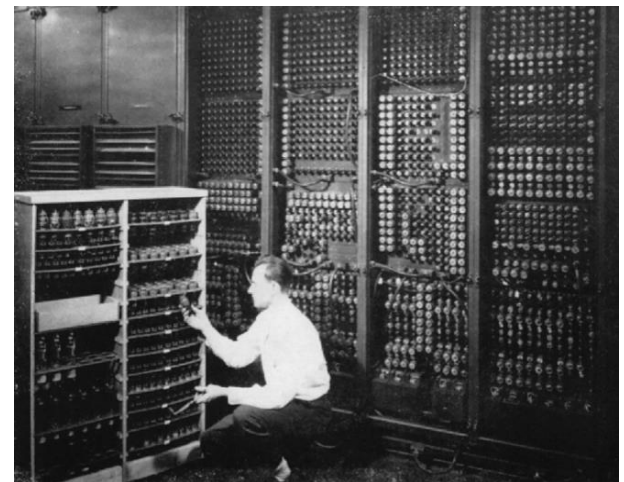
- COLOSSUS and ENIAC were not general purpose computers
  - could enter input using dials & knobs, paper tape
  - but to perform a different computation, needed to reconfigure

von Neumann popularized the idea of a "stored program" computer

- Memory stores both data and programs
- Central Processing Unit (CPU) executes by loading program instructions from memory and executing them in sequence
- Input/Output devices allow for interaction with the user

virtually all modern machines follow this
von Neumann Architecture
    (note: same basic design as Babbage)
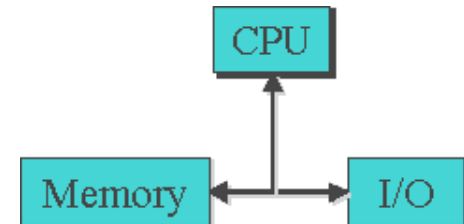
programming was still difficult and tedious

- each machine had its own machine language, 0's & 1's corresponding to the settings of physical components
- in 1950's, assembly languages replaced 0's & 1's with mnemonic names
    - e.g., `ADD` instead of `00101110`

Generation 2: Transistors

- mid 1950's – transistors began to replace tubes
  - a transistor is a piece of silicon whose conductivity can be turned on and off using an electric current
  - they performed the same switching function of vacuum tubes, but were smaller, faster, more reliable, and cheaper to mass produce
  - invented by Bardeen, Brattain, & Shockley in 1948 (earning them the 1956 Nobel Prize in physics)

  some historians claim the transistor was the most important invention of the 20th century

computers became commercial as cost dropped
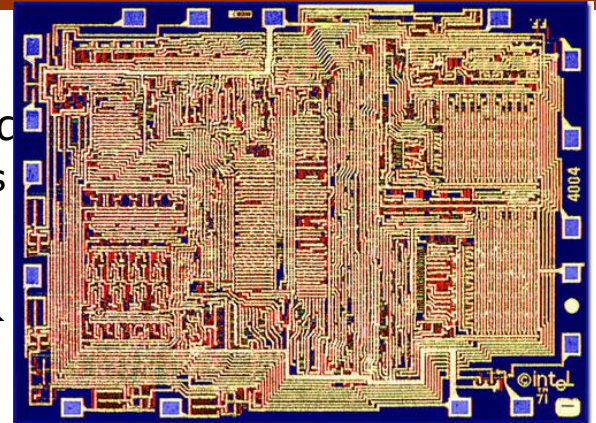high-level languages were designed to make programming more natural

- FORTRAN (1957, Backus at IBM)
- LISP (1959, McCarthy at MIT)
- BASIC (1959, Kemeny at Dartmouth)
- COBOL (1960, Murray-Hopper at DOD)

the computer industry grew as businesses could afford to buy and use computers
   Eckert-Mauchly (1951), DEC (1957)
   IBM became market force in 1960's

Generation 3: Integrated Circuits

- mid 1960's - integrated circuits (IC) were produced
    - Noyce and Kilby independently developed techniques for packaging transistors and circuitry on a silicon chip (Kilby won the 2000 Nobel Prize in physics)
    - this advance was made possible by miniaturization & improved manufacturing
    - allowed for mass-producing useful circuitry

    1971 – Intel marketed the first *microprocessor*, the 4004, a chip with all the circuitry for a calculator
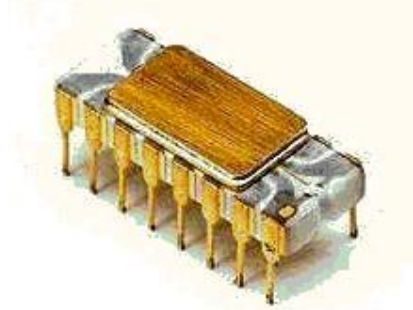
1960's saw the rise of Operating Systems
- recall: an operating system is a collection of programs that manage peripheral devices and other resources
- in the 60's, operating systems enabled time-sharing, where users share a computer by swapping jobs in and out
- as computers became affordable to small businesses, specialized programming languages were developed
    Pascal (1971, Wirth), C (1972, Ritchie)

Generation 4: VLSI

- late 1970's - Very Large Scale Integration (VLSI)
  - by the late 1970's, manufacturing advances allowed placing hundreds of thousands of transistors w/ circuitry on a chip
  - this "very large scale integration" resulted in mass-produced microprocessors and other useful IC's
  - since computers could be constructed by simply connecting powerful IC's and peripheral devices, they were easier to make and more affordable

Year	Intel Processor	Number of Transistors[4]
2009	Quad Core Itanium	2,000,000,000
2006	Core 2 Duo	291,000,000
2000	Pentium 4	42,000,000
1999	Pentium III	9,500,000
1997	Pentium II	7,500,000
1993	Pentium	3,100,000
1989	80486	1,200,000
1985	80386	275,000
1982	80286	134,000
1978	8088	29,000
1974	8080	6,000
1972	8008	3,500
1971	4004	2,300

Generation 4: VLSI (cont.)

with VLSI came the rise of personal computing

- 1975 - Bill Gates & Paul Allen founded Microsoft

  Gates wrote a BASIC interpreter for the first PC (Altair)

- 1977 - Steve Wozniak & Steve Jobs founded Apple

  went from Jobs' garage to $120 million in sales by 1980

- 1980 - IBM introduced PC

  Microsoft licensed the DOS operating system to IBM

- 1984 - Apple countered with Macintosh

  introduced the modern GUI-based OS (which was mostly developed at Xerox)

- 1985 - Microsoft countered with Windows

1980's - object-oriented programming began

- represented a new approach to program design which views a program as a collection of interacting software objects that model real-world entities
- Smalltalk (Kay, 1980), C++ (Stroustrup, 1985), Java (Sun, 1995)

Generation 5: Parallelism/Networks

- the latest generation of computers is still hotly debated
  - no new switching technologies, but changes in usage have occurred

- parallel processing has become widespread
  - multi-core processors provide simple parallelism, can spread jobs across cores
  - similarly, high-end machines (e.g. Web servers) can have multiple CPU's
  - in 1997, highly parallel Deep Blue beat Kasparov in a chess match

Year	Computers on the Internet[5]	Web Servers on on the Internet[6]
2010	758,081,484	205,368,103
2008	570,937,778	175,480,931
2006	439,286,364	88,166,395
2004	285,139,107	52,131,889
2002	162,128,493	33,082,657
2000	93,047,785	18,169,498
1998	36,739,000	4,279,000
1996	12,881,000	300,000
1994	3,212,000	3,000
1992	992,000	50

(Internet Software Consortium & Netcraft, April 2010.)

most computers today are networked
- the Internet traces its roots to the 1969 ARPANet
  - mainly used by government & universities until the   late 80s/early 90s

- the Web was invented by Tim Berners-Lee in 1989, to allow physics researchers to share data
  - 1993: Marc Andreessen & Eric Bina developed Mosaic
  - 1994: Andreesen & Netscape released Navigator
  - 1995: Microsoft released Internet Explorer

- in 2009, 55% of American adults connected to Internet wirelessly, >30% using a smart phone

Evolution of Programming Languages

- late 40's / early 50's: programmers coded directly in machine language

  - each machine had its own set of instructions (sequences of 0's & 1's) corresponding to its underlying hardware

  - → extremely tedious, error-prone

Evolution of Programming Languages

- mid 1950's: assembly languages replaced numeric codes with mnemonic names

  - an *assembler* is a program that translates assembly code into machine code

    - *input*: assembly language program
    - *output*: machine language program

  - still low-level & machine-specific, but easier to program

```
gcc2_compiled.:
        .global _Q_qtod
.section        ".rodata"
        .align 8
.LLC0:  .asciz  "Hello world!"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc    04
main:   !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        sethi %hi(cout),%o1
        or %o1,%lo(cout),%o0
        sethi %hi(.LLC0),%o2
        or %o2,%lo(.LLC0),%o1
        call __ls__7ostreamPCc,0
        nop
        mov %o0,%l0
        mov %l0,%o0
        sethi %hi(endl__FR7ostream),%o2
        or %o2,%lo(endl__FR7ostream),%o1
        call __ls__7ostreamPFR7ostream_R7ostream,0
        nop
        mov 0,%i0
        b .LL230
        nop
.LL230: ret
        restore
```

Evolution of Programming Languages

- late 1950's – present:
- high-level languages allow the
- programmer to think at a
- higher-level of abstraction

- a *compiler* is a program that translates
-     high-level code into machine code

  - *input*: C language program
  - *output*: machine language program

  - similar to assembler, but more complex

- an interpreter is a program that reads and executes each language statement in sequence

  - Python programs are first compiled into a virtual machine language (bytecode)
  - then the bytecode is executed by an interpreter (Python Virtual Machine)

```
/* Hello World in C */

#include<stdio.h>

main() {
  printf("Hello World");
}
```

```
def HelloWorld():
  """ Simple Python function that displays
      a message """
  print "Hello World!"

HelloWorld()
```
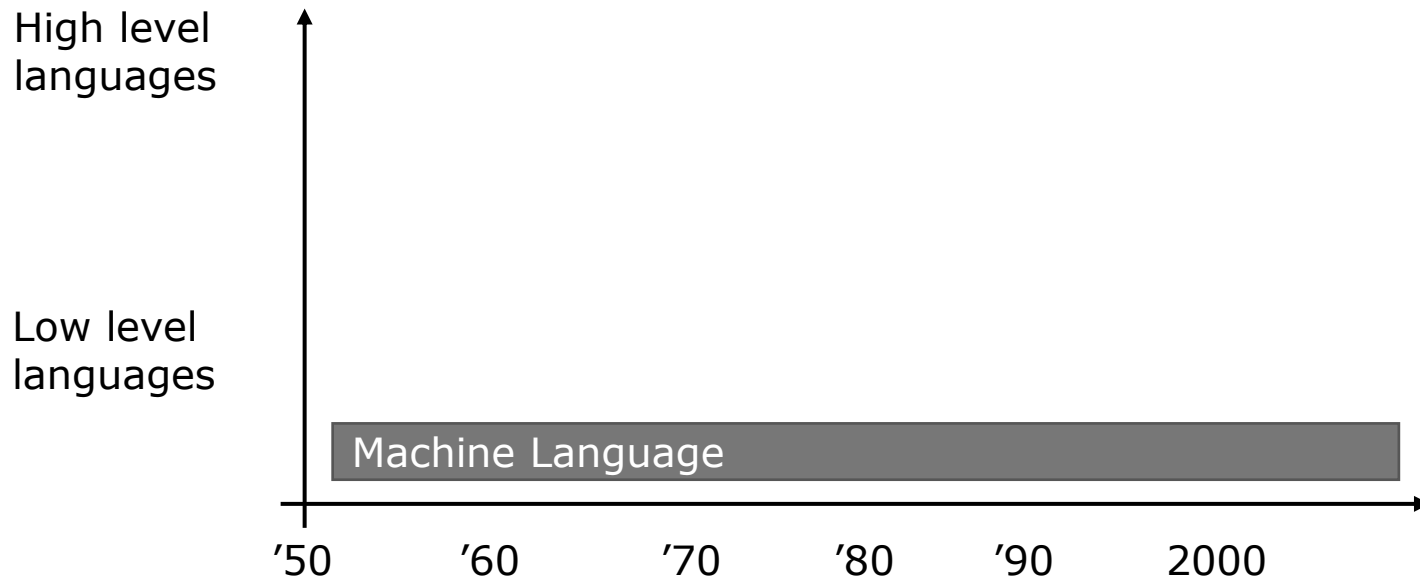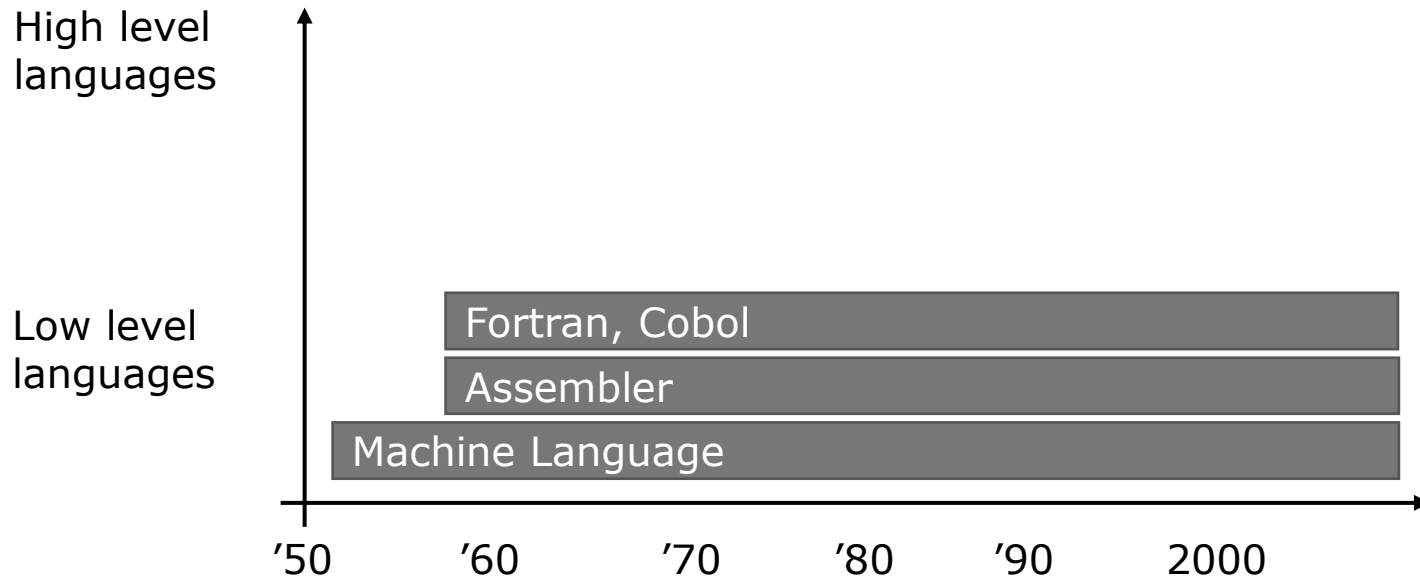
Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)

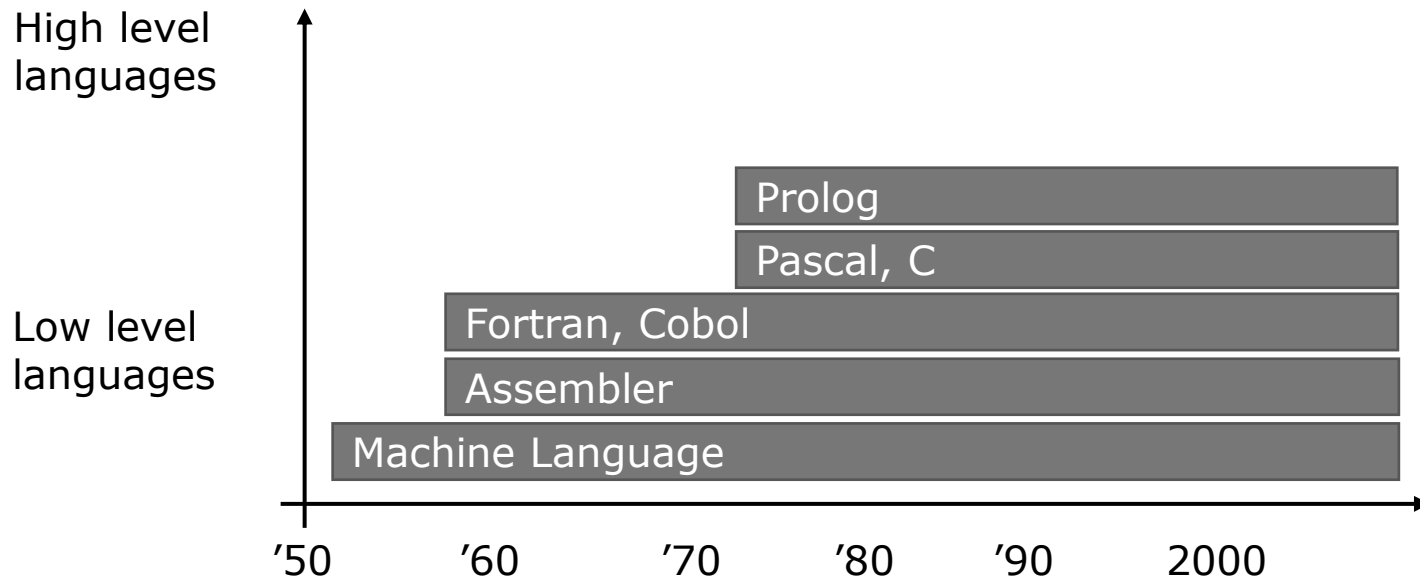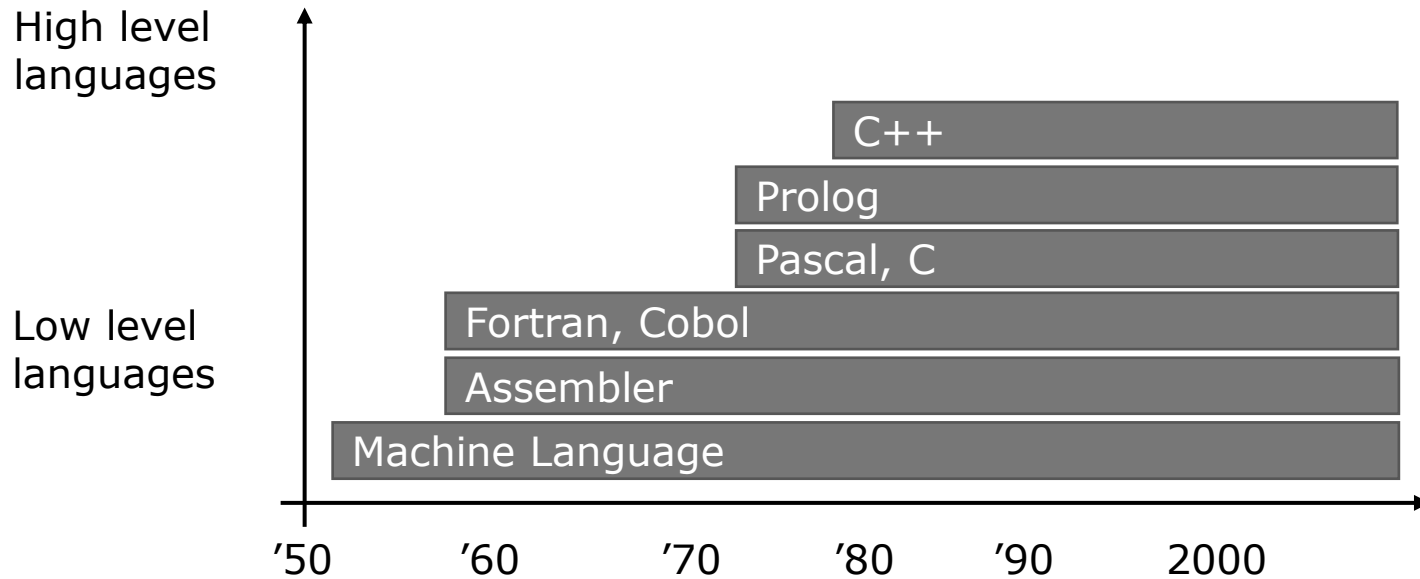| Language	Year	Principal usage	Paradigm
Fortran	1954	Numerical computations	Imperative
Cobol	1959	General	Imperative
Pascal	1971	General	Imperative
Prolog	1972	Artificial Intelligence	Logic
C	1974	General, system programming	Imperative
C++	1979	General	Object Oriented
Python	1991	General	Object Oriented
Java	1995	General	Object Oriented

Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)

High level
languages

Low level
languages

'50   '60   '70   '80   '90   2000

Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)
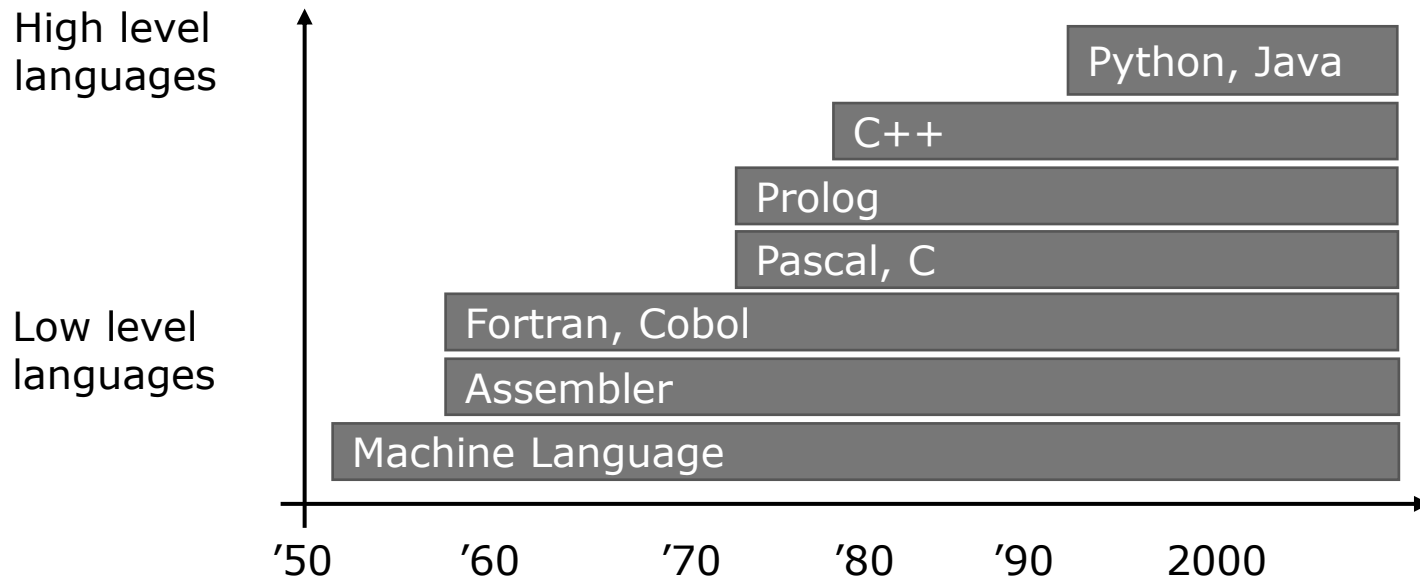
Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)

Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)

Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)

Evolution of Programming Languages

Several programming languages have been proposed since 1950. They can be grouped in different ways (e.g., paradigm, level of abstraction, purpose, etc.)

Evolution of Programming Languages

Low level languages

Early programmers could only use the so-called **Machine Language**: sequenceso of 0s and 1s.

Evolution of Programming Languages

Low level languages

Early programmers could only use the so-called **Machine Language**: sequenceso of 0s and 1s.

The level of abstraction has been augmented with the introduction of the **Assembly language**, in which the instructions (sequences of 0s and 1s) were encoded with symbolic names (es., mov, add)

Evolution of Programming Languages

Low level languages

Early programmers could only use the so-called **Machine Language**: sequenceso of 0s and 1s.

The level of abstraction has been augmented with the introduction of the **Assembly language**, in which the instructions (sequences of 0s and 1s) were encoded with symbolic names (es., mov, add)

The only available operations with such languages are: load values into registers, basic arithmetic, compare values, move to a specific line of code.

Evolution of Programming Languages

High level languages

Exploit keywords in english, which help the programmer to undersand and remember the language instructions.

Evolution of Programming Languages

High level languages

Exploit keywords in english, which help the programmer to undersand and remember the language instructions.

The program must be then processed by an **interpreter** or a **compiler** which translates it into a program in machine language (interpretable by the machine).

Evolution of Programming Languages

High level languages

Exploit keywords in english, which help the programmer to undersand and remember the language instructions.

The program must be then processed by an **interpreter** or a **compiler** which translates it into a program in machine language (interpretable by the machine).

Rapid developing and debugging, however the code is less efficient and requires more resources (e.g., memory, environment tools, etc.)

Languages

- Some influential ones:
  - FORTRAN
    - science / engineering
  - COBOL
    - business data
  - LISP
    - logic and AI
  - BASIC
    - a simple language
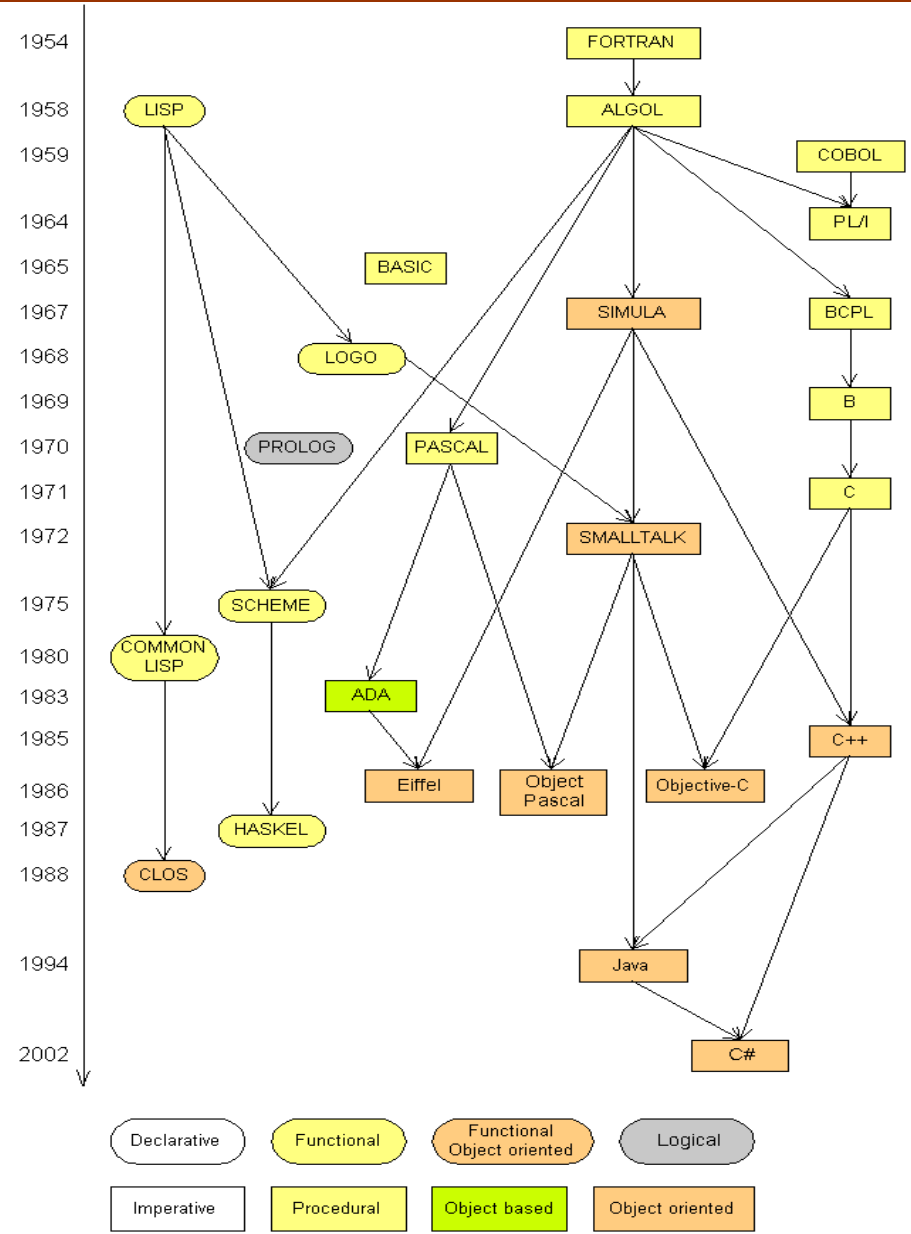
If you want to know more…

- check out the following (purely optional) links

[Inventors: The History of Computers](#)

[Computer Museum History Center](#)

[Transistorized! from PBS.org](#)

[Apple Computer Reading List](#)

[The History of Microsoft](#)

[Internet Pioneers: Tim Berners-Lee](#)

[Internet Pioneers: Marc Andreessen](#)

[Wikipedia entry on Programming Languages](#)
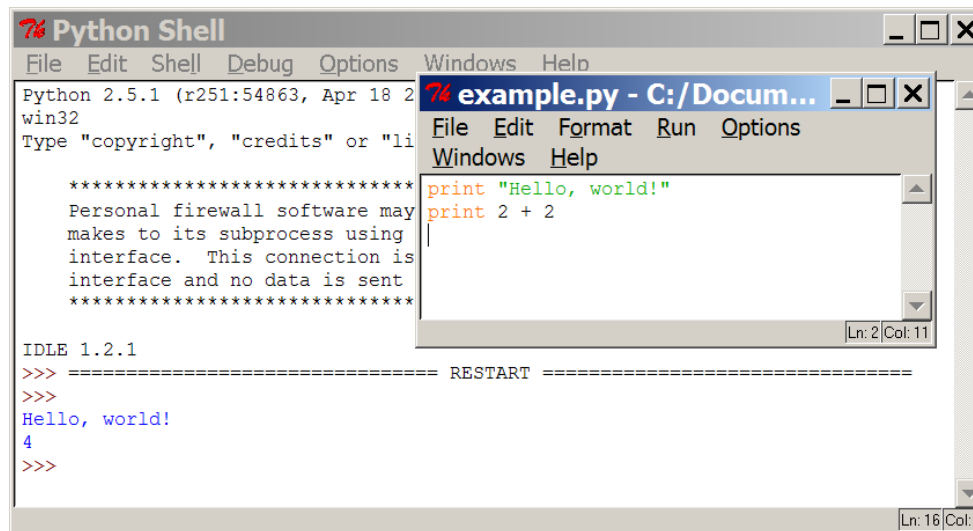
[Webopedia entry on Programming Languages](#)

[Python Official Website](#)

Programming basics

- **code** or **source code**: The sequence of instructions in a program.

- **syntax**: The set of legal structures and commands that can be used in a particular programming language.

- **output**: The messages printed to the user by a program.

- **console**: The text box onto which output is printed.
  - Some source code editors pop up the console as an external window, and others contain their own console window.
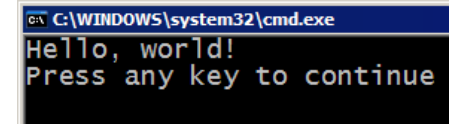
Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.
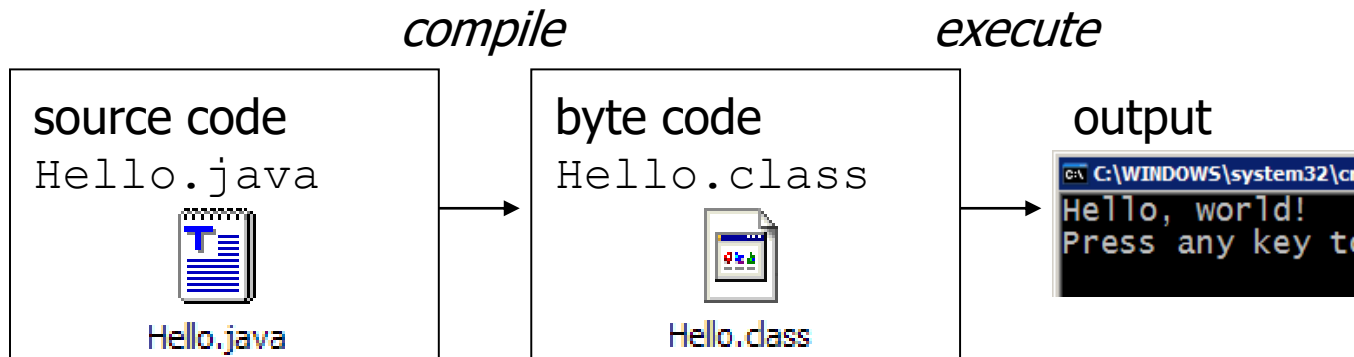
- Python is instead directly *interpreted* into machine instructions.
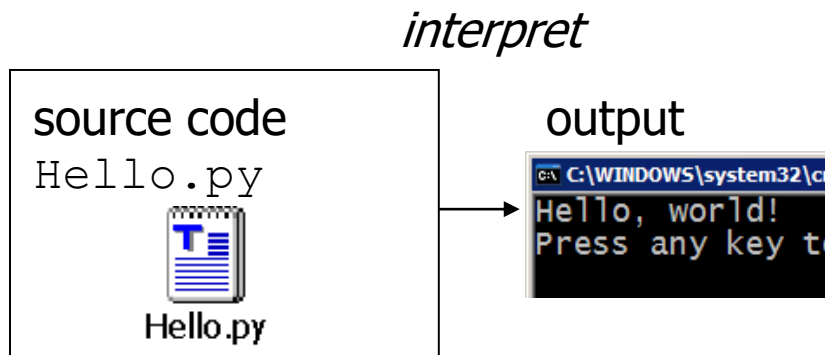
Expressions

- **expression**: A data value or set of operations to compute a value.
  Examples: `1 + 4 * 3`

                     `42`

- Arithmetic operators we will use:
  `+ - * /`   addition, subtraction/negation, multiplication, division
  `%`   modulus, a.k.a. remainder
  `**`   exponentiation

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

    `1 + 3 * 4` is `13`

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is `16`

Integer division

- When we divide integers with `/` , the quotient is also an integer.

```
        3                          52
  4 )   14                  27 )   1425
        12                         135
         2                          75
                                    54
                                    21
```

  - More examples:
    - `35 / 5` is `7`
    - `84 / 10` is `8`
    - `156 / 100` is `1`

- The `%` operator computes the remainder from a division of integers.

```
        3                          43
  4 )   14                   5 )   218
        12                         20
         2                         18
                                   15
                                    3
```

Computer Programming Basics

Computer Programming Basics

- Computer programs are a detailed set of instructions given to the computer

- They tell the computer:
    1. What actions you want the computer to perform
    2. The order those actions should happen in

- An effective program therefore needs:
    1. A thorough understanding of the problem
    2. A well thought-out, step-by-step solution to the problem

HOW TO EAT A BANANA:
A DETAILED LIST OF INSTRUCTIONS

1. Using your hand, get a yellow crescent-shaped fruit called a "banana"
2. Peel the outer skin off the banana (by breaking off the outer stem and peeling back the yellow peel)
3. Eat the banana
    1. Put a small section of banana in your mouth
    2. Bite down on the banana
    3. Chew the banana by opening and closing your mouth
    4. Once the banana has been chewed, swallow the banana
    5. Repeat until banana is finished
4. Throw out the used banana peel

Computer Programming Basics

- A procedure that outlines
  - What actions you want the computer to perform

  and

  - The order in which they happen

  is called an **ALGORITHM**

- An **ALGORITHM** is basically an outline for how your computer program will work

Computer Programming Basics

- Developing an Algorithm is really just a type of Problem Solving
  - We have to:
    - READ and understand the problem
    - THINK about different solutions to the problem
    - DESIGN an approach that will solve the problem
    - IMPLEMENT that design
    - TEST to see if it works

IF NEEDED

Computer Programming Basics

- THINKING about the solution often means breaking down complex tasks into smaller, easier to understand tasks

- These tasks must be **well-defined** so that we understand what the action is acting on
  - e.g. telling a person to grab a banana will only work if the person knows what a banana is

- The tasks have to be **easy to understand**
  - e.g. telling a person to PEEL a banana will only work if they understand what peeling means

Computer Programming Basics

- The ORDER in which actions are performed is also very important

- Consider the following 2 algorithms that tell Mr. Solomon how to get ready in the morning

  We'll call them the "Rise-and-Shine Algorithms"

Computer Programming Basics

RISE AND SHINE 1
- Get out of bed
- Take off pyjamas
- Take a shower
- Get dressed
- Eat breakfast
- Drive to school

RESULT
- Mr. Solomon arrives in a great mood ready to teach ☺

RISE AND SHINE 2
- Get out of bed
- Take off pyjamas
- Get dressed
- Take a shower
- Eat breakfast
- Drive to school

RESULT
- Mr. Solomon arrives in not too great a mood since he's soaking wet ☹

Computer Programming Basics

- When the algorithm is written out as a well-thought series of steps, it is sometimes called **PSEUDOCODE**

- It is written in easy to understand language, but is written very similar to the way that you would code it into your Python Scripts

Computer Programming Basics

- The algorithm can also be written as a FLOW CHART

- The FLOW CHART is a graphic organiser (a picture that helps organize your thoughts)

- It uses a collection of basic symbols that are used to organize your algorithm

- These symbols are connected by arrows that show how the algorithm "flows"

Basics
FLOW CHART SYMBOLS

TERMINAL – the beginning or ending of a program

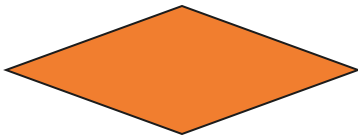INPUT/OUTPUT – where the user of the program is asked for information (INPUT) or where the program displays a result (OUTPUT)
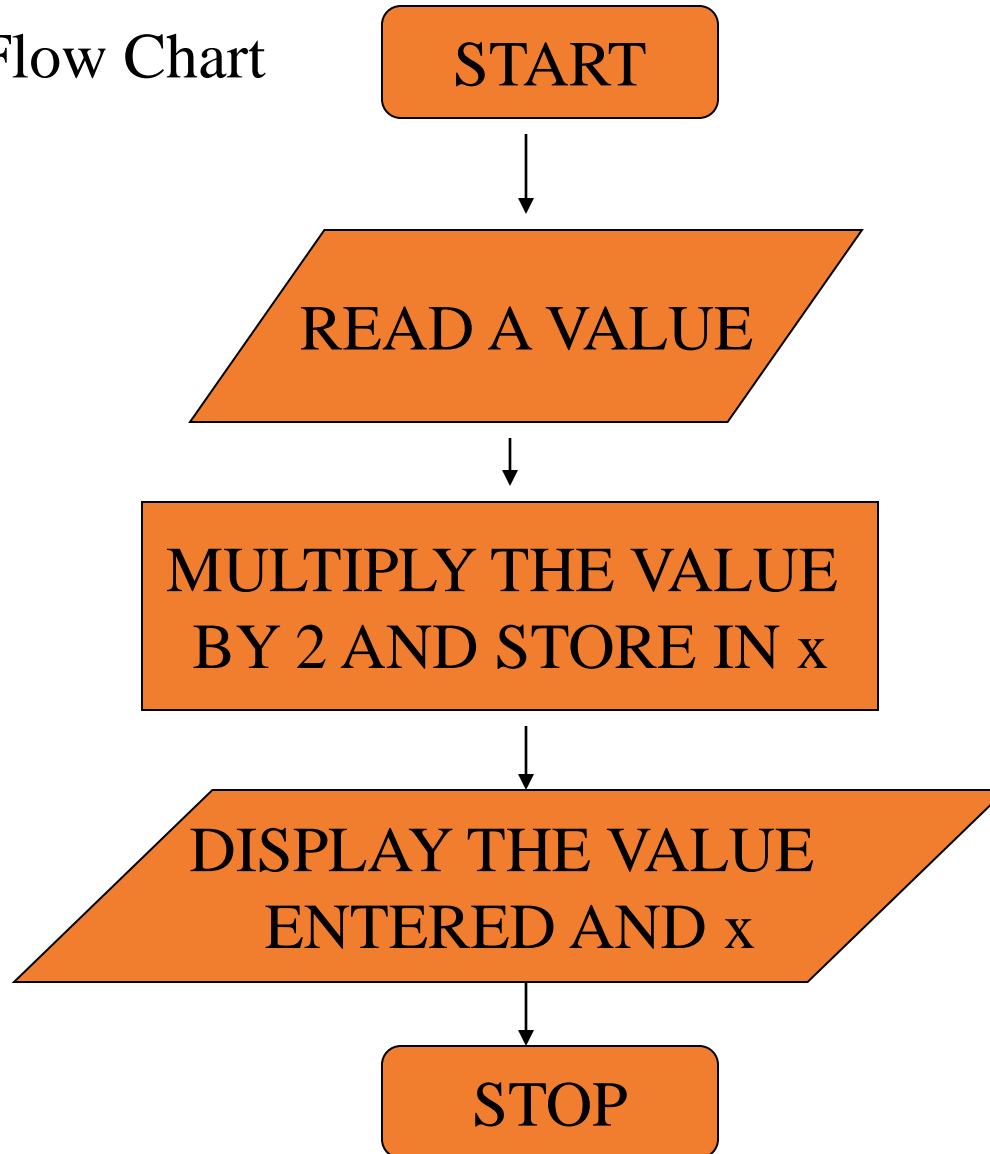
PROCESSING – shows any mathematical operation

CALL – shows any other pieces of the program that are called upon

DECISION – represents any action where the computer is making a decision

Computer Programming Basics

e.g. a Basic Flow Chart

START

READ A VALUE

MULTIPLY THE VALUE BY 2 AND STORE IN x

DISPLAY THE VALUE ENTERED AND x

STOP

Computer Programming Basics

START

↓

GET OUT OF BED

↓

TAKE OFF PYJAMAS

↓

TAKE A SHOWER

↓

GET DRESSED

↓

EAT BREAKFAST

↓

DRIVE TO SCHOOL

↓

END

Examples

Flow Chart Example 1

We will now draw a flow chart for having a bath.

We start by thinking about the steps involved:
(1) Fill the bath with water
(2) Get undressed.
(3) Get into bath.
(4) Wash.
(5) Get out of bath.
(6) Empty the bath water.
(7) Get dressed.

Now we need to draw the chart with instruction boxes. There are no decisions on this chart - the steps all follow on from one another. Remember the **Start** and **End** boxes.

The final chart is shown on the right. Of course some people might do some of these steps in a different order, but hopefully they get undressed before getting in the bath!
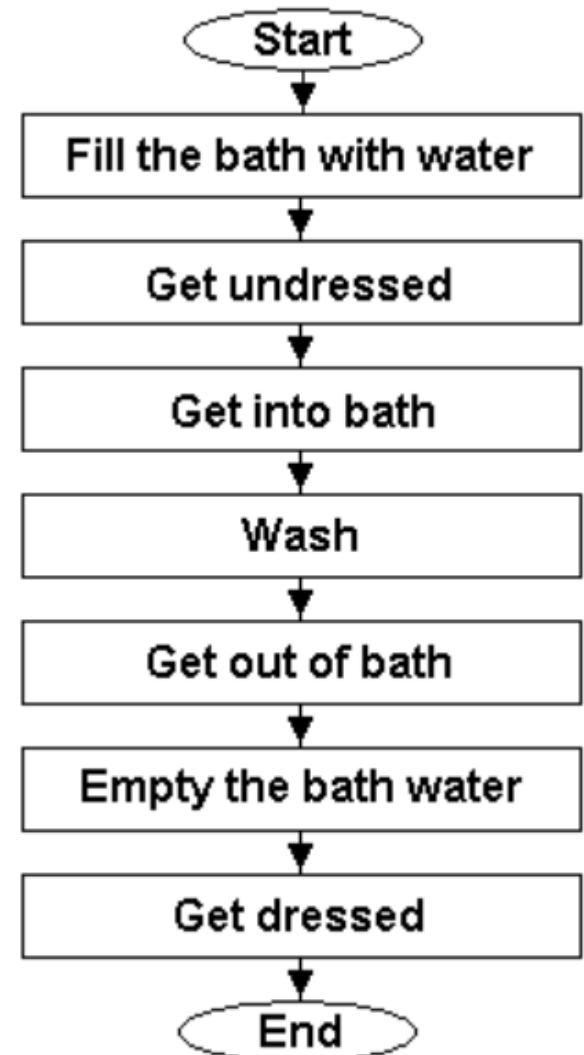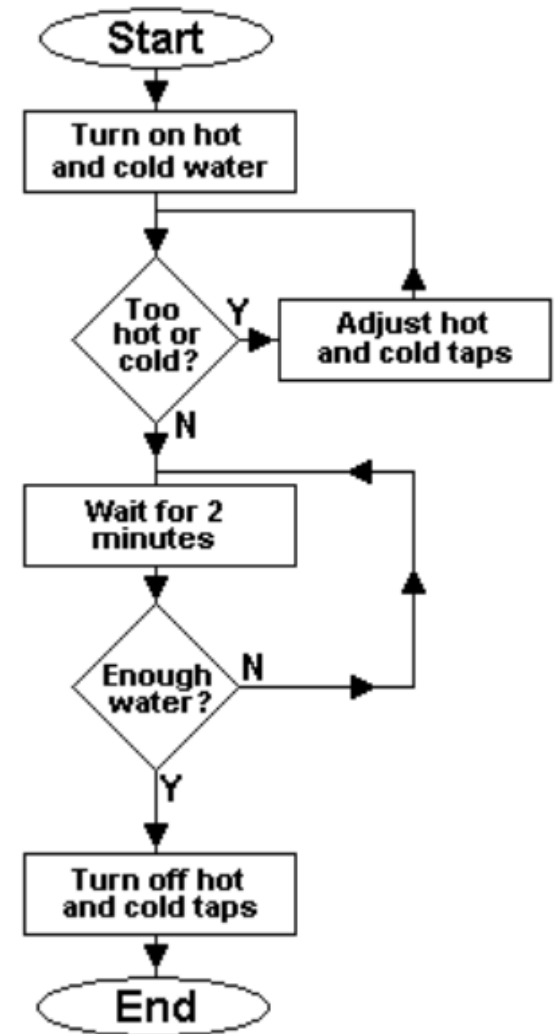
Examples

Flow Chart Example 2

The step *Fill the bath with water* in the previous example could have been more detailed. For example, you need to check if there is enough water and whether it is at the right temperature while running the bath.

Again we need to think about the steps involved:
(1) Turn on the hot and cold taps.
(2) Is it too hot or cold? If it is, go to step 3, otherwise go to step 4.
(3) Adjust the hot and cold taps and go back to step 2.
(4) Wait for 2 minutes.
(5) Is the bath full? If it is, go to step 7, otherwise go to step 6.
(6) Go back to step 4.
(7) Turn off the hot and cold taps.

Now we need to draw the chart. This time we need to use decision boxes for steps 2 (where the temperature of the water is checked) and 5 (where it is checked if the bath is full).

The final chart is shown on the right.

```
        ┌─────────┐
        │  Start  │
        └────┬────┘
             ▼
     ┌──────────────┐
     │ Turn on hot  │
     │ and cold water│
     └──────┬───────┘
            ▼
         ◇ Too          Y   ┌──────────────┐
         hot or  ──────────▶│ Adjust hot   │
          cold?             │ and cold taps│
            │ N             └──────────────┘
            ▼
     ┌──────────────┐
     │ Wait for 2   │◀────────┐
     │  minutes     │
     └──────┬───────┘
            ▼
        ◇ Enough    N
          water?  ──────────┐
            │ Y
            ▼
     ┌──────────────┐
     │ Turn off hot │
     │ and cold taps│
     └──────┬───────┘
            ▼
        ┌─────────┐
        │   End   │
        └─────────┘
```

Exercise

Question

The flow chart on the right is meant to show the steps for stopping working on a computer and shutting it down.

Place the instructions below in the flow chart. Some of the instructions are not required - you should only include those which are relevant to the task.

Quit the program	Check your electronic mail
Switch off the machine	Turn on the computer
Finish working on your document	Select 'shut down'
Start a new document	Save your work on a disk

Start

End

Exercise

Question

The flow chart on the right is meant to show the steps for stopping working on a computer and shutting it down.

Place the instructions below in the flow chart.
Some of the instructions are not required - you should only include those which are relevant to the task.

Quit the program	Check your electronic mail
Switch off the machine	Turn on the computer
Finish working on your document	Select 'shut down'
Start a new document	Save your work on a disk

Start
↓
Finish working on your document
↓
Save your work on a disk
↓
Quit the program
↓
Select 'shut down'
↓
Switch off the machine
↓
End

Variables

- **variable**: A named piece of memory that can store a value.
  - Usage:
    - Compute an expression's result,
    - store that result into a variable,
    - and use that variable later in the program.

- **assignment statement**: Stores a value into a variable.
  - Syntax:

    name = *value*

  - Examples:
    ```
    x = 5
    gpa = 3.14
    ```

    x | 5 | gpa | 3.14 |

  - A variable that has been given a value can be used in expressions.
    `x + 4` is `9`

- **Exercise:** Evaluate the quadratic equation for a given *a*, *b*, and *c*.

Variable names

- some reserved words cannot be used for variable names

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

Python libraries tend to use underscores for multi-word variable names

```
first_name     number_of_sides     feet_to_meters
```

we will utilize the more modern (and preferred) camelback style

```
firstName     numberOfSides     feetToMeters
```

note: capitalization matters, so `firstName ≠ firstname`

Variables & assignments

- Variables are used to store values that could be accessed & updated
    - e.g., the lines spoken by the sprites in conversation
    - e.g., the number of spins and player bankroll for slots

- In Python can create a variable and assign it a value
    - the variable name must start with a letter, consist of letters, digits & underscores (note: no spaces allowed)
    - an assignment statement uses '=' to assign a value to a variable
    - general form: `VARIABLE = VALUE_OR_EXPRESSION`

        - `age = 20`
        - `secondsInDay = 24 * 60 * 60`
        - `secondsInYear = 365 * secondsInDay`
        - `name = "Prudence"`
        - `greeting = "Howdy " + name`

Number types

- Python distinguishes between different types of numbers
  - **int**   integer values, e.g.,   2, -10, 1024, 99999999999999
    .

    - ints can be specified in octal or hexidecimal bases using '0o' and '0x' prefixes
    - $0o23 \rightarrow 23_8 \rightarrow 19_{10}$ $0x1A \rightarrow 1A_{16} \rightarrow 26_{10}$

  - **float**   floating point (real) values, e.g.,   3.14, -2.0, 1.9999999

    scientific notation can be used to make very small/large values clearer
    $1.234e2 \rightarrow 1.234 \times 10^2 \rightarrow 123.4$ $9e-5 \rightarrow 9 \times 10^{-5} \rightarrow 0.00009$

  - **complex**   complex numbers (WE WILL IGNORE)

Real numbers

- Python can also manipulate real numbers.
  - Examples: `6.022`   `-15.9997`   `42.0`   `2.143e17`

- The operators `+ - * / % **  ( )` all work for real numbers.
  - The `/` produces an exact answer: `15.0 / 2.0` is **7.5**
  - The same rules of precedence also apply to real numbers: Evaluate `( )` before `* / %` before `+ -`

- When integers and reals are mixed, the result is a real number.
  - Example: `1 / 2.0` is `0.5`

  - The conversion occurs on a per-operator basis.
    ```
    7 / 3 * 1.2 + 3 / 2
      2   * 1.2 + 3 / 2
        2.4     + 3 / 2
        2.4     +   1
             3.4
    ```

Numbers & expressions

- standard numeric operators are provided

+	addition	`2+3` → `5 2+3.5` → `5.5`
−	subtraction	`10−2` → `8` `99−99.5` → `−0.5`
*	multiplication	`2*10` → `20` `2*0.5` → `1.0`
/	division	`10/2.5` → `4.0` `10/3` → `3.333…`
**	exponent	`2**10` → `1024` `9**0.5` → `3.0`

- less common but sometimes useful

%	remainder	`10%3` → `1` `10.5%2` → `0.5`
//	integer division	`10//4.0` → `2.5` `10//4` → `2`

Math commands

- Python has useful [commands](#) for performing calculations.

Command name	Description
abs(*value*)	absolute value
ceil(*value*)	rounds up
cos(*value*)	cosine, in radians
floor(*value*)	rounds down
log(*value*)	logarithm, base *e*
log10(*value*)	logarithm, base 10
max(*value1*, *value2*)	larger of two values
min(*value1*, *value2*)	smaller of two values
round(*value*)	nearest whole number
sin(*value*)	sine, in radians
sqrt(*value*)	square root

Constant	Description
e	2.7182818…
pi	3.1415926…

- To use many of these commands, you must write the following at the top of your Python program:
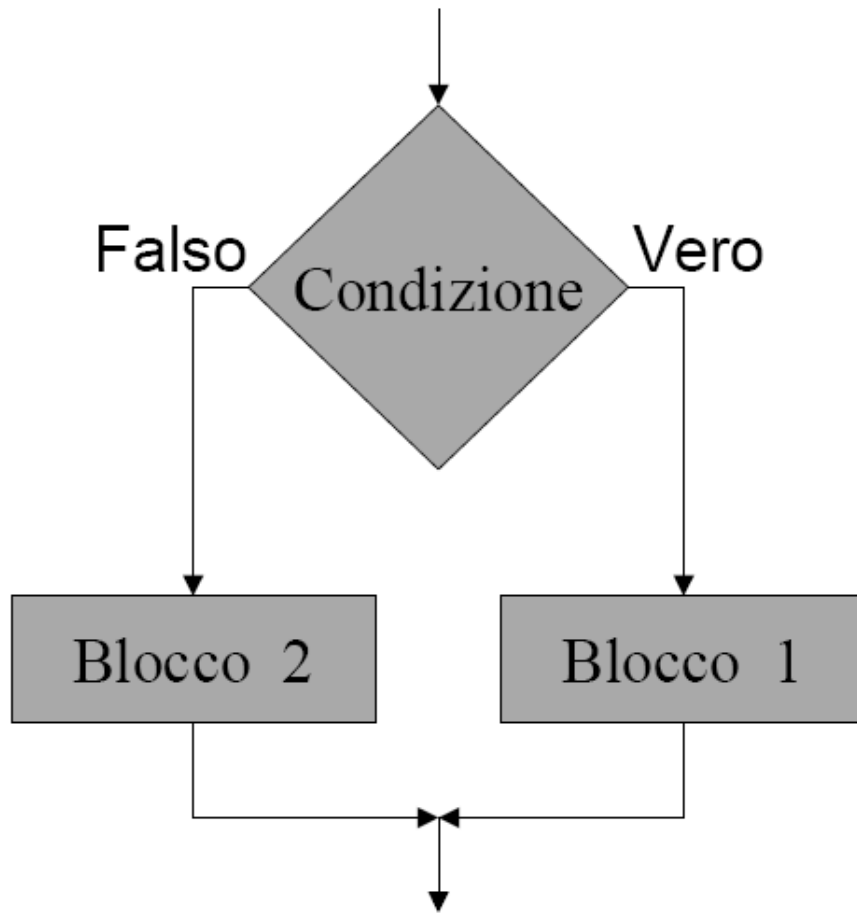
```
from math import *
```

Istruzione 1

Istruzione 2

...

Istruzione n

$\equiv$

Istruzione 1;

Istruzione 2;

...

Istruzione n;

Falso   Condizione   Vero

Blocco

$\equiv$
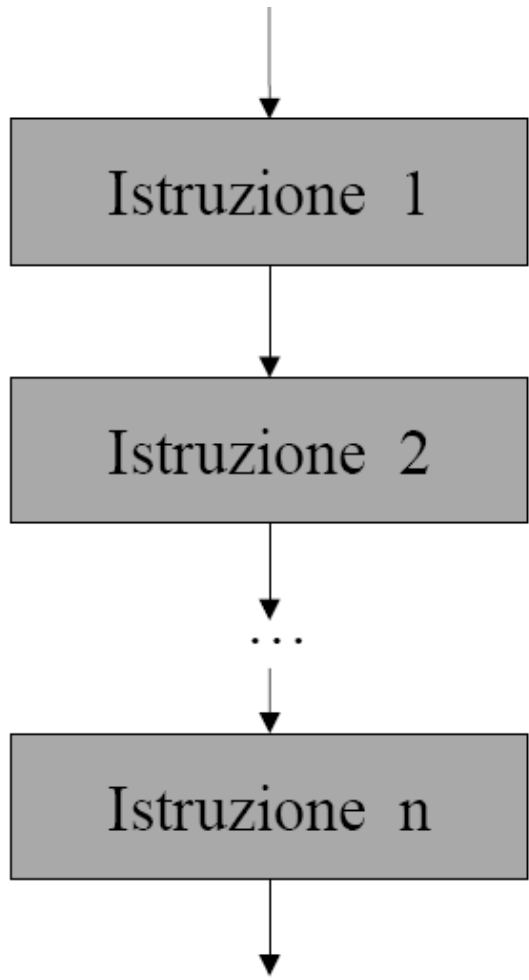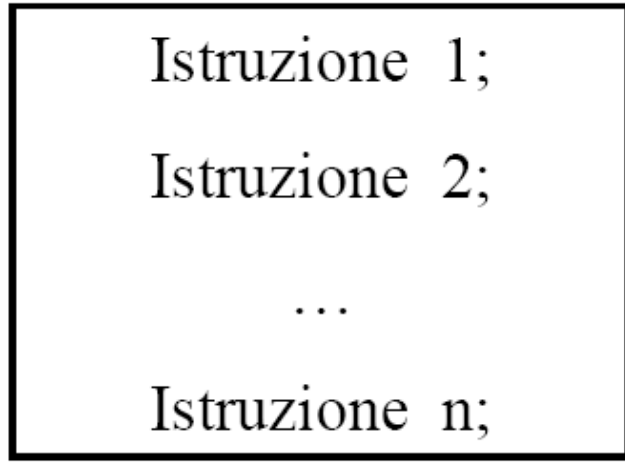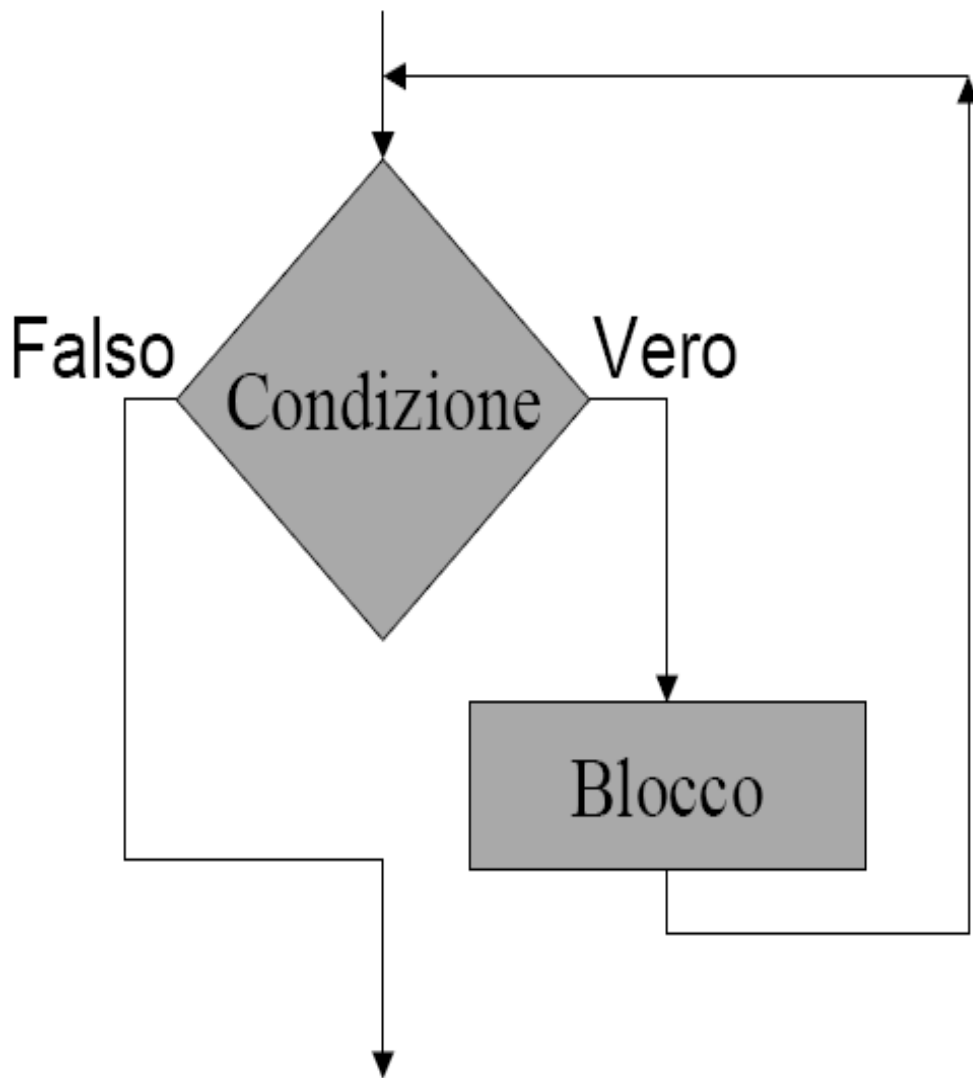
while (Condizione) do

Blocco;

end while;

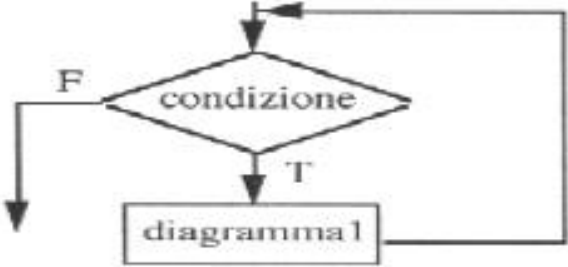Falso — Condizione — Vero

Blocco 2   Blocco 1

≡
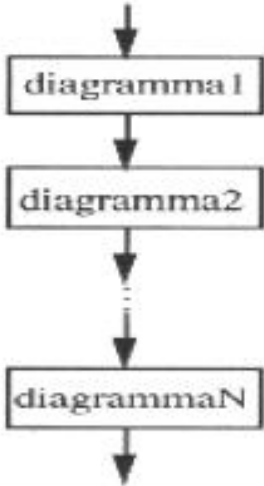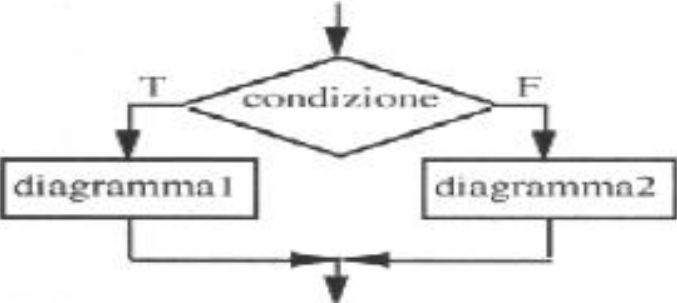
if   (Condizione) **then**

       Blocco 1;

 else

       Blocco 2;

end if

La clausola "else" può anche essere assente.

diagramma	significato
diagramma1 → diagramma2 → diagrammaN	Diagramma di sequenza
T ← condizione → F, diagramma1, diagramma2	Diagramma condizionale
condizione (F, T) → diagramma1	Diagramma di ripetizione

Exercises

- Given an integer number *n*:
    1. Is it odd/even?
    2. Is a multiple of *k*?
    3. Is it a prime number?
- Given a series of *N* integer numbers compute:
    1. The overall sum (product);
    2. The average;
    3. The minimum and maximum value;
    4. To **count** the number of elements greater (lesser) than a number *k*
    5. ….

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377...

1+1=2	13+21=34
1+2=3	21+34=55
2+3=5	34+55=89
3+5=8	55+89=144
5+8=13	89+144=233
8+13=21	144+233=377

$$f(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ F(n-1) + F(n-2) & if \quad n > 1 \end{cases}$$

Factorial

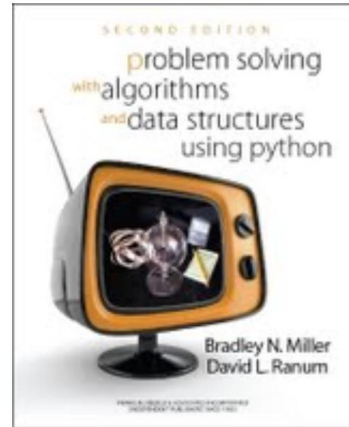$$0! = 1$$
$$1! = 1$$
$$2! = 1 \cdot 2 = 2$$
$$3! = 1 \cdot 2 \cdot 3 = 6$$
$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$
$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$
$$6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$$

Problem Solving with Algorithms and Data Structures using Python

By Brad Miller and David Ranum, Luther College

- Assignments

- 1. Introduction
    - 1.1. Objectives
    - 1.2. Getting Started
    - 1.3. What Is Computer Science?
    - 1.4. What Is Programming?

- https://runestone.academy/runestone/books/published/pythonds/index.html