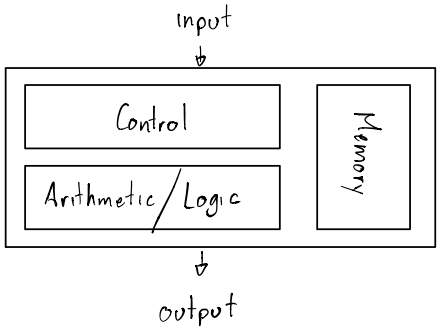
¿What is computing?

- Computer used to be a profession, a career
- Ada Lovelace proposed the Analytical Engine
- Alan Turing solved all algorithms with a system of data manipulation.
- Binary system... Easily represented by electronic signals
- Von Neumann makes EDVAC, Computer with memory
- Richard Feymann → Quantum computing

Computación actual



Lenguajes de programación

- Declarative: Relationships between variables
- Imperative: It's about a route to a goal... algorithms → Finite list of instructions

- C is like latin
- Python tries to be more easy to read/write

Python !

Understandable by computers — Low level vs High level — Understandable by humans

Broadly applicable, lacks specialized features — General vs. Specialized

Executes line by line — Interpreted vs. Compiled — Are converted to machine code and then executed. Usually more efficient and gives more control

Syntax

< variables > or < objects > : 1 (int), 'abc' (string), true (boolean), 1.0 (float)

< operators >:

+	-	*	**	/	//	%	
plus	minus	times	power	division	floor division	modulus	Arithmetic

==	!=	<>	<	>	<=	>=	
equal	not equal		less than	greater than	less than or equal to	greater than or equal to	Comparison

set value / reset value → = Assign

x < arithmetic operator > = y

is equivalent to

x = x < arithmetic operator > y

Ex. x += y is equivalent to x = x + y

Logical	Bitwise
and or not	& | ^ ~ << >>

< object > < operator > < object > = < expression >

Print statement: print (< expression >)

Ex. 5 / "platzi" ... semantic error

Type error

because syntax is ok

Object

Values in memory that can be referenced

method `type()` → `type("name") = string`

Clean code Name variables with relevant words

Reserved words

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

Strings

- Character chain
- Between ' ' or " "

Usual operations

- Concatenate: - "str_1" + "str_2" = "str_1str_2"
  - f'Hello world {"!" * 3} = "Hello world !!!"
- Length: len('Hello') = 5
- Indexing: my_str = "Hello" → my_str[2] = "l"
- Slicing: my_str[:2] = "He"
  my_str[2:] = "llo"
  my_str[::2] = "Hlo"

Floats

An approximation...

Because numbers must be converted to binary code, there can be weird situations with floats. Ex.

$$0.1 \times 10 = 0.99999...$$

It is better to use > or < instead of == with floats

Inputs

```
nombre = input("Cual es tu nombre?")
```

Conditionals

and or not

```
if <condition>:
    <expresion>
elif <condition2>:
    <expresion2>
else:
    <expresion3>
```

Truth table

A	B	A and B	A or B
T	F	F	T
T	T	T	T
F	T	F	T
F	F	F	F

Loops

while for iter

break to get out of the loop prematurely

Enumerate

Every possibility... brute force

Binary search

Must have ordered assets to work with.
- Cuts the array in half and uses the half where the object it is looking for is located. Over an over again until found.

If you are looking for number 8 in a 9-number array: STEPS.

① 1 2 3 4 5 / 6 7 8 9 — uses this half

② 6 7 / 8 9 — uses this half

③ 8 9 — find

ABSTRACTION - Use of libraries

Decomposition - Divide the code in functions

Functions

Keyword

```
def <name> (<parameters>):
    <body>
    return <expresion>
```

- Can have a default value (not mandatory to use)
- Can be declared on call

Scope and frames

Scope Python

Es el contexto al que pertenece una variable en el programa de Python

¿En qué partes del programa puedo usar una variable?

Scope global

Si creas una variable en el cuerpo principal del código de Python, puedes usarla en cualquier parte

```
x = 'Esta es una variable global'
def funcion():
    print(x)

funcion()

print(x)
```

Puedes crear una variable local con el mismo nombre de una variable global, pero Python lo leerá como 2 variables distintas

```
x = 'Esta es una variable global'
def funcion():
    x = 'Y esta es una variable local'
    print(x)

funcion()

print(x)
```

Scope local

Si creas una variable dentro de una función, esta pertenece sólo a esa función

```
def funcion():
    x = 'esta es una variable local'
    print(x)

funcion()
```

También puedes usar la variable en una función dentro de otra función

```
def funcion():
    x = 'esta es una variable local'
    def funcion_interior():
        print(x)
    funcion_interior()

funcion()
```

Keyword global

Puedes crear una variable global dentro de una función usando el keyword global

```
def funcion():
    global x
    x = 'Esta es un variable global'

funcion()

print(x)
```

También puedes cambiar el valor de una variable global con este keyword

```
x = 'Esta es una variable global'
def funcion():
    global x
    x = 'Y ahora cambiamos su valor'

funcion()

print(x)
```

```python
def func1(un_arg, una_func):
    def func2(otro_arg):
        return otro_arg * 2

    valor = func2(un_arg)
    return una_func(valor)

un_arg = 1

def cualquier_func(cualquier_arg):
    return cualquier_arg + 5

func1(un_arg, cualquier_func)
```

Contexts

Global

func1
un_arg
cualquier_func

func1

un_arg
una_func
func2
valor

func2

otro_arg

cualquier_func

cualquier_arg
un_arg

Docstrings

- Documentation with:
  - What the function does
  - Parameters used by the function
  - What the function returns

"q" To quit docstrings in terminal

Recursion

- "Divide and conquer"
- Call to itself

Ex. Factorial $n! = \prod_{i=1}^{n} i$ → Normal loop

$n! = n \cdot (n-1)!$ → Recursive

Factorial definition inside Factorial definition

Ex. $3! = 3 \cdot 2!$ → $3! = 3 \cdot 2 \cdot 1$

$2! = 2 \cdot 1!$

$1! = 1$

Python recursion limit

Danelia Sanchez Sanchez Estudiante · el año pasado

Desconocía totalmente que Python tuviera un límite de recursividad. Para conocerlo hay que importar la librería sys:

```
>>> import sys
>>> print(sys.getrecursionlimit())
1000
```

Para modificar ese límite

```
sys.setrecursionlimit(n) # n es el nuevo límite a establecer
```

https://www.geeksforgeeks.org/python-sys-setrecursionlimit-method/

Funciones como objetos

Las funciones:
- Tienen tipo
- Se pueden pasar como argumentos de otras funciones
- Se pueden utilizar en expresiones <función> == "hola"
- Se pueden incluir en varias estructuras de datos

Tuples

- Immutable sequence of objects → Can be reassigned
- Can be used with return statement return (1, 2)

- Defined with parenthesis unlike lists

```
my_tuple = (1, 2)
my_list = [1, 2]
```

- Can use an specific value ... my_tuple[0]
- Define one element tuple with a comma... my_tuple = (1,)
- Unpack to variables... x, y, z = my_tuple
  - Must have same amount of elements as variables

Ranges

- integer sequence with start, finish and pace.
- Unmutable

```
range(0, 10, 2)
      start  finish - NOT INCLUDED
             pace
```

id() method to get objects id

```
range(0, 7, 2) == range(0, 8, 2)   → Value equality
id(range1) is id(range2)           → Object equality
```

False!

```
For i in range(0, 101, 2):
    print(i)
```
→ List every odd number from 0 to 100

Lists

- Mutable
- Modify lists may cause side effects
- Iterable

Functions: Modify → my_list[0] = 5 → Assign value to a place

- Append
- Pop
- Remove
- Insert
- Remove
- Extend
- Copy
- Clear
- Index
- Count
- Sort
- Copy
- Reverse

Clone a list

`list(a)` or `b = a[::]`

List comprehension

- Use operations along a list
- Conditions to filter

Documentation

https://docs.python.org/3/tutorial/datastructures.html

Dictionaries

- Lists with keys instead of index
- No order
- Hashmaps — Hashing function
- Mutable
- Iterable

```
my_dict = { key_1: value_1, key_2: value_2, ... }
```

Tests

Black box tests

- It's based on method description, not implementation
- Try inputs validating outputs
- Unit testing or integration testing

class, test driven development

Crystal box tests

- Program flow
- Test possible paths
- Regression testing or mocks

if - Test if, elif, esle

Recursions & loop - No loop
- Just once
- A lot of times

while - Break conditions
- False entry conditions

Debugging

- Print statement > debugger
- Scientific method
- Open mind
- Keep in mind what you've tested

search... Binary search with print statements

Exceptions

- Code errors, common errors
- Defensive programming
- Key words: 1. Try
  2. Except
  3. Finally
- Don't silence exceptions
- Send exceptions with "raise"

EAFP

Easier ask for forgiveness than permission

```
try:
    return pairs[pair]
except KeyError:
    return None
```

vs. Verifying the existance of "pair" before

Affirmations

- verify object types

assert <boolean>, <error message>