

Introducción al Pensamiento

Computacional con Python

Contenido

- Resolución de problemas de manera computacional
- Técnicas desarrolladas en los últimos 50 años
- Bases para carrera en Computer Science
- Multilenguaje: temas en común para todos los lenguajes de programación
- Piezas fundamentales de los lenguajes de programación

Módulo I:

von Neumann \Rightarrow genera estructura del computador moderno.

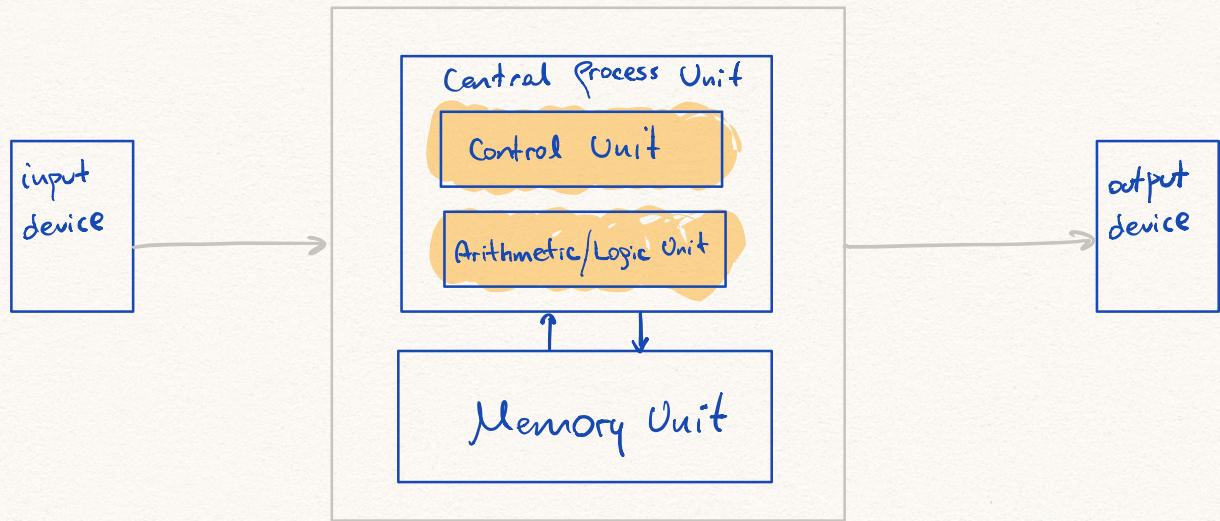
nube \Rightarrow data centers.

interesante: Richard Feynman

dio las bases matemáticas para crear sistemas de cómputo cuántico

FUTURO
estas computadoras se están comenzando a crear

Arquitectura de von Neumann



Computadora \Rightarrow hace 2 cosas

hacer cálculos recordar el resultado de esos cálculos

se empieza a romper problema de velocidad del cerebro y la mano

A la computadora se le da instrucciones:

> Conocimiento declarativo: nos dice qué tipo de relación existe entre diversas variables.

> Conocimiento imperativo: nos dice cómo llegar a un resultado



Algoritmos: primeros programas de computación para el primer sistema de cómputo (cerebro)

“lista finita de instrucciones que describen un cálculo, que cuando se ejecuta con ciertas entradas (inputs) ejecuta pasos intermedios para llegar a un resultado (output).”

C \Rightarrow el "latín" de los lenguajes de la programación

python \Rightarrow el lenguaje "más naturalizado" para el ser humano.

Lenguajes

todos tienen:

- sintaxis: define una secuencia de símbolos bien formada
- semántica estática: define qué enunciados con sintaxis correcta tienen significado.
- semántica: define el significado.

En los lenguajes de programación sólo hay un significado.

Módulo II: Elementos básicos de Python (pero aplicable a cualquier lenguaje de programación)

Lenguajes de Programación

Bajo nivel: más parecido a unos y ceros

General: posee todos los "primitivos" de Turing para computar cualquier algoritmo

Interpretado: se traduce el lenguaje en tiempo de ejecución

Alto nivel: más parecido al lenguaje humano

Domínio específico: lenguajes especializados a aplicaciones muy específicas.

Compilado: se traduce a lenguaje máquina antes de la ejecución

python \Rightarrow alto nivel, general, interpretado.

Elementos básicos:

Literales: 1, 'abc', 2.0, True

operadores: + / * % ** = ==

estos interactúan, se
interpolan



<literal> <operador> <literal>
<objetos> <operador> <objetos>

1

+

2

'hola'

+

'mundo'

↓
esto es una "expresión"

Objetos \Rightarrow son la abstracción más alta en cualquier lenguaje de programación.

"así se modula el mundo en una computadora"

distintos tipos: desde un Int, hasta conjuntos de valores

escalares: vs no escalares
- subdividible - no subdividible
(ejemplo: vector) (ejemplo: integer)

Variable: nombre vinculado a valor en memoria

son ayudas para comprender los algoritmos

Es IMPORTANTE asignar nombres con significado a las variables.

* ('Hip' * 3) + '' + 'hurra'

||

f'{ "Hip" * 3 } hurra'



cadena más legible de Python

NOTA:

en python, los strings no son alterables.

función input () \Rightarrow recibe strings que retorna el usuario.

¿cómo formateo a int esa string recibida?



valor_entero = int (input('Dame un valor'))

Programas ramificados (3 formas)

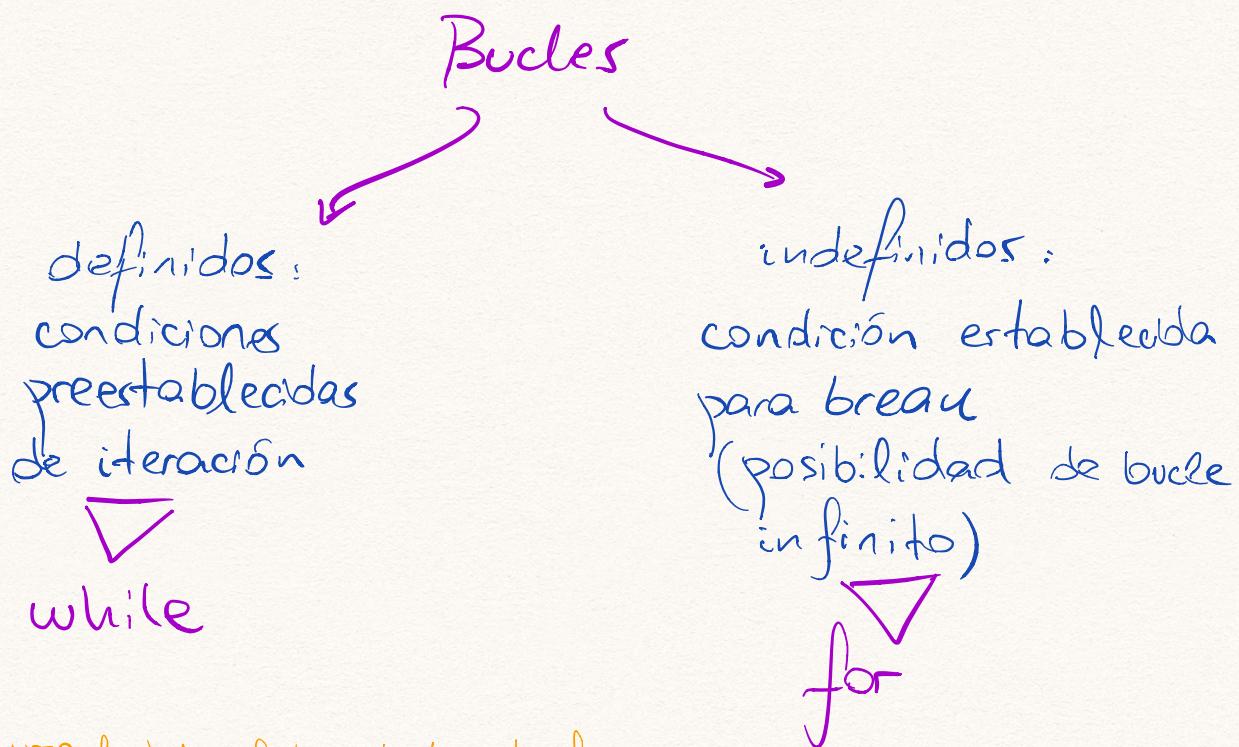
① if <condition>:
 <expression>

② if <condition>:
 <expression>
else:
 <expression>

③ if <condition>:
 <expression>
elif:
 <expression>
else:
 <expression>

Iteraciones (Loops):

- se usa para que un programa haga lo mismo varias veces.
- las iteraciones se quedan aúdiar.
- se puede utilizar break (a caso de python) para salir anticipadamente de un loop (ejemplo: si se cumple una condición).
- OJO: loops infinitos.



for en Python:

for <variable> in <iterable>:
 <expression>

strings

lists

tuples

arrays

dictionaries

↓
todos son iterables.

NOTA: Ejemplos varios y a detalle de loops: Clase 11 de este curso.

Palabras para alterar los loops: break y
continue

floats (decimales) ↴

- * Precavición con los float; los números binarios presentan limitaciones para con los números decimales. Es por esto que a veces no son perfectos.
Al no ser perfectos, debemos verificar la igualdad de los floats de manera distinta al resto:

Igualación normal: $x == 1$

Igualación de floats:

$$x < 1.0 \text{ and } x > 0.99999$$

Módulo III : Programas numéricos

Algoritmo "enumeración exhaustiva"

- ↳ uno de los primeros algoritmos que debes tratar
- ↳ también llamado "adivina y verifica"
- ↳ no es eficiente en manejo de recursos pero, al ser las computadoras actuales tan rápidas, no importa en términos prácticos.

Algoritmo "aproximación de soluciones"

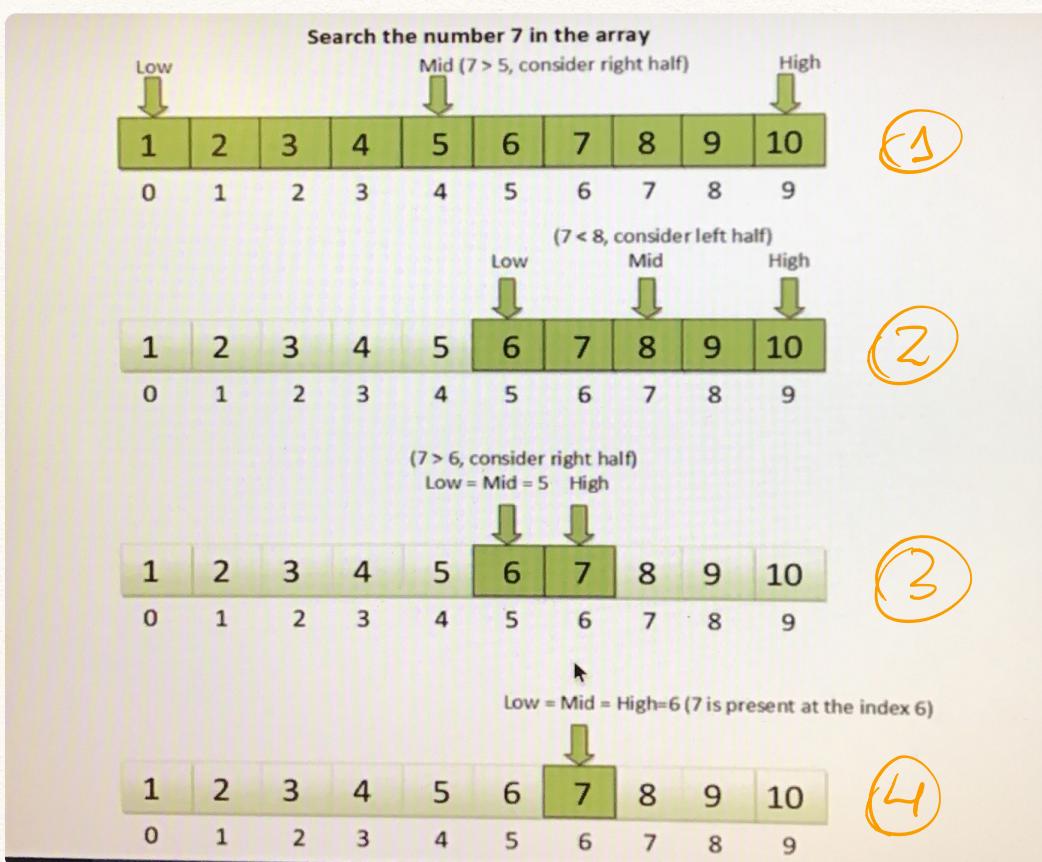
- ↳ similar a enumeración exhaustiva
- ↳ no necesita respuesta exacta
- ↳ acepta un margen de error
 - ↳ se le llama epsilon
 - ↳ cuanto más preciso es epsilon (menos margen de error, más recursos serán necesarios)
 - trade off; es situacional.

Algoritmo "búsqueda binaria"

↳ cuando la respuesta se encuentra en un conjunto ordenado, podemos usar la búsqueda binaria.

↳ altamente eficiente, pues corta el espacio de búsqueda en dos por cada iteración

↳ algoritmo de los más importantes de computer science.



Búsqueda de N° 7

- ① Paso 1: partimos conjunto en dos mitades; sólo nos quedamos con la mitad que contenga 7 ($S > 7$? No, entonces nos quedamos con conjunto posterior a S)
- ② Iterar paso 1
- ③ Iterar paso 1
- ④ Resultado

Comparado a búsqueda exhaustiva, el ahorro de recursos es enorme.

Módulo IV : Funciones y alcance de las funciones

* Hasta, eramos capaces de hacer código spaghetti (todo el código en un mismo archivo, ilegible)



para mejorar esto, los lenguajes modernos han creado nuevas herramientas, como lo son las funciones, scope (alcance) de las mismas y demás.

CONCEPTOS CLAVE

Abstracción: no se necesita entender la forma en que funciona internamente algo, para poderlo usar.

Ejemplo: usas calculadora sin saber funcionamiento interno.



es el ingeniero de software quien crea estas abstracciones (desde una librería, API, a un programa más complejo)

Decomposición:

- ↳ dividir el código en componentes que colaboran con un fin en común.
- ↳ "mini programas" que son componentes de un programa mayor.

Funciones:

- permite abstractar mi código
- permite decomponer mi código

```
def nombre_funcion (parámetros):  
    código_implementado  
    return resultado_de_código
```

NOTA: python toma la indentación para detectar la estructura de la función. En otros lenguajes esto puede variar, pero la lógica y estructura de una función, es la misma.

Las funciones pueden requerir parámetros requeridos de forma obligatoria, o pueden también tener parámetros opcionales con valores por defecto.

```
def nombre_completo (nom, apellido, inverso=False)
```

valor por defecto

CONCEPTO IMPORTANTE; conveniente ahondar en el tema

Scope (Alcance): cada función es un contexto particular dentro de un programa.
El Scope marca a cuáles contextos tiene acceso una función.

DocStrings: instrucciones y documentación que explica qué hace tal o cual parte del código. En python, las docstrings comienzan y terminan con triple comilla ("")

tiene 3 partes:

- qué hace
- qué significan parámetros
- qué retorna (regresa)

¿Cómo accedo a los docstrings de una función?



help(function)

BUENA PRÁCTICA: escribir los docstrings antes de desarrollar el código; da guía, ayuda a concretar la idea y no perder el foco.

Recursividad (Importante)

↳ Algorítmica: es una forma de crear soluciones utilizando el principio de "divide y vencerás"



↳ dividir problema grande en problemas pequeños

↳ Programática: técnica mediante la cual una función se llama a sí misma.

↳ Es un loop hacia la misma función

(*) Tema de necesaria profundización

Funciones como objeto

→ característica particular de python, muy potente.

En python, todo es un objeto, incluso
las funciones

las funciones:

- tienen un tipo
- se pueden pasar como argumentos en otras funciones
- se pueden usar en expresiones \star^1
- se pueden incluir en varias estructuras de datos (lists, tuples, dictionaries, etc)

también llamadas
funciones anónimas

esto abre las puertas a aplicaciones mucho más complejas

\star^1 : las expresiones se aplican usando la keyword lambda

(lambda <vars> : <expression>)

Ejemplo: `sumar = lambda x,y: x + y`
`sumar(2,3) => 5`

Módulo V: Tipos estructurados, mutabilidad y funciones de alto nivel

Tuplas (tuples):

- ↳ secuencias inmutables de objetos
«Esto es, una lista de valores que NO podemos modificar
- ↳ pueden contener cualquier tipo de objeto (a diferencia de las cadenas)
- ↳ puede utilizarse para devolver varios valores en una función
 «return [x,y]

tuple-1 = (1,)

tuple-2 = (2,3,4)

tuple-1 += tuple-2 \Rightarrow print (tuple-1) } suma de tuplas
(1, 2, 3, 4)

x, y, z = tuple-2 \Rightarrow {
 x = 2
 y = 3
 z = 4 } desempaquetado de tuplas

Rangos:

- ↳ representan una secuencia de enteros
- ↳ range (comienzo, fin, paso)
- ↳ los rangos son inmutables (al igual que los strings)
- ↳ son muy eficientes en uso de memoria y normalmente utilizados en for loops.

NOTA : value equality \Rightarrow se compara con ==
object equality \Rightarrow se compara con is

Listas (estas sí son mutables):

- ↳ secuencias de objetos mutables
- ↳ cuidado con alterar listas (side effects)
- ↳ es posible iterar con ellas.

tenemos :

$$a = [1, 2, 3]$$

b = a \Rightarrow apuntan al MISMO espacio en memoria!!

si modificamos b, modificamos a.
Cuidado.

Clonación (solución a side effects)

↳ casi siempre es mejor clonar una lista, que mutarla

↳ se clona con slice o list

↳ ejemplo:

```
a = [1, 2, 3]
c = list(a)
c = a[::]
```

} esto genera
otro espacio en
memoria (clon)

slice ↗ recomendable akondar en documentación.

List comprehension

↳ forma concisa de aplicar operaciones a los valores de una secuencia.

↳ es posible aplicar condiciones para filtrar

Diccionarios (dictionary)

- ↳ son como listas, pero en lugar de utilizar indices, utilizan llaves (keys)
- ↳ no tienen orden interno
- ↳ son mutables
- ↳ se pueden iterar
- ↳ se obtiene un acceso super eficiente.

Módulo VII: Gestión de errores (bugs)

Pruebas de caja negra.

- ↳ se basan en la especificación de la función o el programa
- ↳ prueba inputs y valida outputs
- ↳ Unit testing o integration testing

↳ Pruebas unitarias:
prueban función por función,
viendo que el código
funciona

er cuando vemos
que todos los módulos
funcionan entre sí.

se trata de generar
un test ANTES de generar
nuestras funciones, de modo
que, a todo momento,
tendremos un respaldo que
nos dirá si nuestro código
funciona o no, incluso en
modificaciones futuras

es aquí donde más valioso es nuestro testing; si volvemos al código (nosotros o un tercero) luego de mucho tiempo, tendremos la protección y alerta necesaria.

Pruebas de caja de cristal

- ↳ se basan en el flujo del programa
- ↳ prueban todos los caminos posibles de una función. Ramificaciones, bucles for y while, recursión.
- ↳ regression testing o mockups

unit test depura la mayoría de los bugs.
→ aun así, siempre habrá un bug que se escape

Debugging → nos suceden a todos
↓
↳ se trata de arreglar bugs, errores

Reglas generales para debugging:

- no molestanse con el debugger. Ayudarse del print statement
- estudia los datos disponibles
- utiliza los datos para crear hipótesis y experimentos (método científico)

- tener mente abierta. "Si entendieras el programa, probablemente no habrían bugs."
- Lleva registro de lo que has tratado, preferentemente en forma de test.

DEBUGGING => LEARN SKILL

IMPORTANTE

Diseño de experimentos para debugging

↳ debuggear es un proceso de búsqueda.
Cada prueba debe acotar el espacio de búsqueda

↳ Búsqueda binaria con print statements.

Errores comunes

- ↳ encuentra sospechosos comunes (la mayoría de los bugs viven allí)
- ↳ en lugar de preguntarte por qué un programa no funciona, pregúntate por qué está funcionando de esta manera
- ↳ es posible que el bug no se encuentre donde crees que está
- ↳ explícale el problema a otra persona, de preferencia que no tenga contexto

- ↳ Lleva registro de lo que has intentado, preferentemente en la forma de tests.
- ↳ Dormir \Rightarrow es importante descansar el cerebro.

Manejo de excepciones: (código defensivo)

- ↳ son muy comunes en la programación
- ↳ se pueden crear excepciones propias
- ↳ si una excepción no se maneja, el programa colapsa.

Keywords (en python)

- ↳ try
- ↳ except
- ↳ finally

↳ excepciones también sirven para ramificar programas

↳ MALA PRÁCTICA: silenciar errores (esto es, mandar error a la consola con print statement)

↳ para "aventar" excepción propia, keyword raise

Excepciones como control de flujo.

↳ aparte del control de errores, python usa las excepciones para el control de flujo.

↳ ¿Para qué si ya existe if, elif, else?

↳ principio EATP de python (easier to ask for forgiveness than permission)

||
↳ para comparar, JavaScript utiliza el principio LBYL (look before you leap, revisa antes de saltar)

código verifica antes los keys, attributes, indexes para ver si son válidos, y luego procede.

↓
asume keys, attributes, indexes válidos; luego los captura en caso de excepción

EAFP (python):

```
def busca_pais(paises, pais):
    """
    Paises es un diccionario. Pais es la llave.
    Codigo con el principio EAFP.
    """

    try:
        return paises[pais]
    except KeyError:
        return None
```

LBYL (JavaScript):

```
// Javascript

/**
 * Paises es un objeto. Pais es la llave.
 * Codigo con el principio LBYL.
 */
function buscaPais(paises, pais) {
    if(!Object.keys(paises).includes(pais)) {
        return null;
    }

    return paises[pais];
}
```

NOTA: LBYL también se puede utilizar en Python, pero no es la filosofía de este lenguaje.

Afirmaciones

- ↳ programación defensiva
 - ↳ mecanismo para determinar si una condición se cumple o no, y poder seguir con la ejecución de nuestro programa, o simplemente no hacerlo.
- ↳ sirven para debugging.
- ↳ corroboran que tipos de parámetros sean correctos para una función.

Keyword: assert

Ejemplo:

assert <condición>, "mensaje error"

```
# assert <expresión booleana>, <mensaje de error>

def primera_letra(lista_de_palabras):
    primeras_letras = []

    for palabra in lista_de_palabras:
        assert type(palabra) == str, f'{palabra} no es str'
        assert len(palabra) > 0, 'No se permiten str vacíos'

        primeras_letras.append(palabra[0])

    return primeras_letras
```

length (largo) de la palabra tiene que ser mayor a 0.

palabra debe ser de tipo string (str)