AA2021a-01-IntroPython

Abdiel E. Cáceres González

28 de diciembre de 2020

Índice

Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.

2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

PARTE UNO
Prerrequisitos

1. Panorama general

Definición 1 (Algoritmo)**.** Un algoritmo es una secuencia finita y ordenada de pasos bien definidos para resolver un problema computacional bien especificado, a partir de insumos determinados.

Un **problema computacional**, es la descripción de un problema que establece en términos generales una relación entre los datos de entrada y la información de salida. El algoritmos describe un procedimiento computacional para lograr la relación entrada/salida deseada.

Ejemplo 1: Problema computacional

Un ejemplo de un problema computacional es el siguiente texto:
Problema: Sumar de los primeros n números enteros.

Entrada: Un número entero positivo n.

Salida: Un número que corresponde a la suma $\sum_{i=1}^{n} i$.

Un algoritmo es similar en algunos aspectos a una receta de cocina, o a procedimientos, en el sentido de que ofrecen una manera detallada de resolver un problema. Pero son diferentes en que los algoritmos son descritos de forma precisa, mientras que las recetas o procedimientos en general, en ocasiones permiten ambigüedad en su descripción. Hay cinco características importantes en cualquier algoritmo:

1. **Finitud.** Los algoritnos deben tener un número finito de pasos. Un algoritmo que tenga un número infinito de pasos, nunca davolverá un resultado, por lo que el algoritmo se vuelve inútil. En ocasiones el número de pasos no es infinito, pero es tan grande que prácticamente es igual como si fuera infinito, digamos por ejemplo un algoritmo que tardará 10,000 años en terminar. Aunque esta cifra parezca ridículamente grande, hay algoritmos que bajo ciertas condiciones pueden tardar esa cantidad de tiempo, o incluso más.

   En otras ocasiones el número de pasos que requiere el algoritmo es infinito, pero se debe lograr alguna condición para detener el proceso. Por ejemplo en algunos algoritmos de optimización, el valor óptimo no se alcanza, o converge muy lentamente, y la solución que ofrece el algoritmo se encuentra en una vecindad cercana al óptimo. El diseño del algoritmo debe considerar una salida de emergencia para tales situaciones.

2. **Especificidad.** Cada paso del algoritmo debe ser especificado con precición. Cada orden que debe ejecutar la computadora, debe establecerse sin dar lugar a confusión alguna. Cuando un algoritmo se escribe en un lenguaje natural [como español] es inevitable algo de ambigüedad, sin embargo espero que no sea tan confuso en el caso de los algoritmos en este curso. Para evitar en lo posible esta situación, se incluyen algunas expresiones en el lenguaje de las matemáticas, especialmente aquellas que son mas conocidas, en otras ocasiones se recurre a un lenguaje de programación, en este caso `Python`.

3. **Entrada.** Un algoritmo es una manera ordenada de resolver alguna tarea, para lograr su cometido el algoritmo [al igual que cualquier otra máquina o proceso] se requieren ciertos *insumos*, que deben ser cumplidos antes de empezar el algoritmo, por ejemplo la suma de dos números, requiere de entrada, dos números. La entrada de un algoritmo normalmente se especifica como una secuencia de cero o más identificadores llamados *parámetros formales* o simplemente *parámetros*. En ocasiones los

parámetros son obligatorios y en otras ocasiones son optativos; cuando los parámetros son optativos, debe haber una manera de estabelcer un valor por defecto para tales parámetros.

4. **Salida.** Un algoritmo tiene también una o más salidas. Una salida es un valor que pudiera ser nulo o no serlo, también puede ser una función.

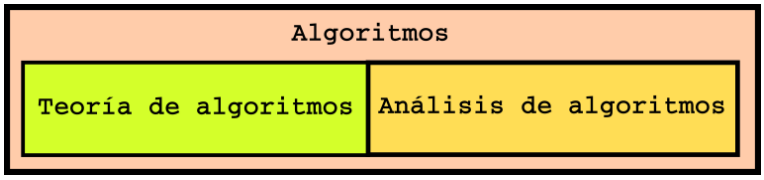
5. **Efectividad.** La utilidad del algoritmo es precisamente porque puede resolver alguna tarea específica, es decir el algoritmo es efectivo para esa tarea. Dentro de la efectividad del algoritmo debe considerarse que el tiempo que toma el algoritmo para realizar la tarea no debe ser infinito, de hecho aún cuando el tiempo empleado sea finito, no debe ser tan grande que prácticamente se considere infinito. El tiempo es la característica que hace que un algoritmo sea mejor que otro al resolver la misma tarea.

En la práctica, no solo queremos algoritmos efectivos, sino algoritmos eficientes, y un buen criterio para determinar una mejor eficiencia en los algoritmos es el tiempo que requieren para devolver un resultado. Frecuentemente se prueban diferentes algoritmos que resuelvan la misma tarea, luego se decide cuál de ellos es mejor en términos del tiempo de ejecución.

La práctica con los algoritmos involucra dos grandes actividades:

1. **La teoría [diseño] de algoritmos**. Esta actividad trata principalmente con determinar la existencia o inexistencia de procedimientos efectivos para resolver tareas computacionales.

2. **El análisis de algoritmos**. Dado un algoritmo, queremos determinar sus características de desempeño, tanto en el gasto de memoria como en el tiempo de ejecución.



Las personas que se dedican a las ciencias computacionales, enfrentan dos retos importantes:

1. Encontar una solución al problema computacional planteado [diseño de algoritmos].

2. Determinar qué tan buena es la solución y posiblemente encontrar una mejor solución [análisis de algoritmos].

Este curso se trata de conocer con más detalle estas dos actividades fundamentales en el área de las ciencias computacionales. Para este curso dedicaremos un poco de tiempo para aprender el lenguaje de programación utilizado en este curso, luego algunas técnicas de análisis de algoritmos y finalmente seis de las técnicas algorítmicas más importantes.

1.1. Diseño de algoritmos

El diseño de algoritmos se trata de bosquejar lo más detalladamente posible, una estrategia para dar solución a un problema específico.

Hay una gran variedad de técnicas de diseño de algoritmos, clasificar las técnicas puede ser una tarea bastante ardua y difícil de llevar a cabo por los diferentes propósitos y enfoques de los problemas que se pretenden resolver con algoritmos, sin embargo ya se han construido algunas técnicas que son frecuentemente utilizadas en el diseño de algoritmos, de modo que tomar una de tales técnicas puede ser un buen comienzo al empezar un diseño.

Técnicas de diseño de algoritmos:

1. **Algoritmos recursivos**. Un algoritmo es recursivo si en alguna parte de la descripción del procedimiento hay al menos una llamada al mismo algoritmo. Esta es una de las técnicas de diseño de algoritmos más elegante, porque describen la solución de un problema en términos de la solución de un caso muy simple, o de un problema con las mismas características pero de un tamaño menor.

2. **Algoritmos divide y vencerás**. [divide and conquer] Este tipo de algoritmos se parece un poco a los algoritmos recursivos en el sentido de que tratan de encontrar una solución, dividiendo el problema original en dos o más subproblemas de estructura y características similares, pero de menor tamaño, y luego haciendo un proceso de reunificación de las soluciones. Hay algoritmos de ordenamiento que utilizan esta técnica, también en problemas de búsquedas, algunos problemas de grafos o de multiplicación de matrices; son solamente algunos ejemplos.

3. **Algoritmos de programación dinámica** [dynamic programming] Cuando el costo espacial [memoria utilizada] es relativamente barato en comparación con el costo temporal, es un indicio para intentar una solución vía programación dinámica. En este tipo de algoritmos se guardan soluciones a problemas de diferentes tamaños, con el fin de que se tenga memoria de una solución previa y ofrecer la solución muy rápidamente. El costo computacional se incrementa cuando no hay registro de un problema con características particulares, entonces el algoritmo resuelve el problema en algún otro modo, pero una vez encontrada esa solución, ya quedará disponible por si en el futuro se requiere de ella.

4. **Algoritmos voraces** [greedy]. Los algoritmos voraces también son conocidos como algoritmos golosos, ávidos o devoradores. La idea subyacente en este tipo de algoritmos es siempre elegir aquella solución que aporte más beneficio, con la esperanza de terminar más rápido y con el mayor monto de beneficio. Normalmente se ocupa en problemas de optimización. Es un tipo de algoritmos que es relativamente fácil de diseñar, de programar y de probar; aunque frecuentemente ocurre lo de aquel refrán «lo barato puede salir caro». Un algoritmo muy conocido que utiliza esta técnica es el algoritmo de Dijkstra para encontrar la ruta más corta entre dos vértices de un grafo.

5. **Algoritmos de fuerza bruta** [brute force] Este tipo de algoritmos son los más seguros y garantizan ofrecer la solución deseada si la hay; y si no hay solución, también es seguro que no qedará lugar sin buscar o posible solución que probar. Los algoritmos de fuerza bruta encuentran una manera de probar cada una de todas las posibles soluciones, sin importar cuál «parezca» mejor, simplemente se sigue un orden en las soluciones, y solamente hasta después de haber probado todas las soluciones, es por eso que es tan seguro. El problema es que frecuentemente encontrar todas las soluciones y probarlas, ocupa tanto tiempo que es mejor utilizar otra técnica algorítmica.

6. **Algoritmos de vuelta atrás** [backtracking] Este tipo de algoritmos son utilizados en problemas de búsquedas. La idea es recorrer todas las soluciones, como los algoritmso voraces, pero el modo de hacerlo es lo interesante, van dejando un rastro y ponen una marca cada vez que tienen más de una solución que probar, exploran una de ellas y siguen así. Cuando es hora de probar una solución pendiente, regresan hasta una marca puesta anteriormente y exploran una nueva ruta de soluciones.

Para diseñar un buena algoritmo es necesario tener conocimiento de diferentes recursos computacionales, en cierto modo es como diseñar construcciones, en la medida que el diseñador conozca diferentes tipos de elementos de construcción, podrá ofrecer soluciones de mejor calidad.

Entre los recursos que es «obligatorio» conocer, son las **estructuras de datos** y algunas técnicas matemáticas que permitirán mejores diseños.

1.2. Análisis del tiempo de ejecución

Una vez que se tiene un algoritmo que resuelve la tarea especificada, se debe realizar un aálisis del tiempo de ejecución, que es el proceso de determinar cómo se incrementa el tiempo que requiere el algoritmo para terminar la tarea, en función del tamaño del problema. El tamaño del problema depende del tipo de problema en cuestión, por ejemplo en la tarea de ordenamiento, el tamaño del problema suele ser el tamaño del arreglo donde estan alojados los números. Otras medidad usuales son:

- El grado de un polinomio,
- El número de elementos en una matriz,
- El tamaño de una palabra,
- El número de bits en la representación binaria de la entrada,
- El número de vértices y aristas en un grafo, etc.

El tiempo de ejecución no suele ser una buena medida para comparar algoritmos, porque el tiempo empleado depende de diferentes factores como:

1. El tipo de computadora en la que se ejecute la tarea,
2. La carga de procesos ocultos que el sistema operativo esté ejecutando en ese momento.
3. El estilo de la programación.

Sin embargo se puede analizar el tiempo de ejecución para lograr una expresión que relacione el tiempo de ejecución como una función del tamaño de la entrada y así comparar diferentes funciones, correspondientes a diferentes tiempos de ejecución. Este tipo de comparación es independiente del tiempo de la computadora, estilo de programación , etc.

La razón en la que el tiempo de ejecución cambia a medida que el tamaño del problema aumenta se llama **razón de crecimiento**, y se presenta en forma de una función matemática.

Cuando se comparan algoritmos en términos de su razón de crecimiento, usualmente se hace omitiendo los detalles menos significativos. Imagina esta situación: Sales de tu casa para hacer algunas compras. Es un día especial porque vas a comprar un auto nuevo de agencia, pero ese mismo dia vas a comprar también una bicicleta y unos patines. De repente te encuentras con un amigo al cual saludas cordialmente pero no quieres dar muchas explicaciones. Ahora imagina el siguiente dialogo:
– ¿Que tal, qué andas haciendo? – saludó el amigo al verte.
– Vine a comprar – respondes apurado.
– ¿Qué vas a comprar? – vuelve a preguntar.
– Un auto – respondes sin más detalle.

Observa que el costo del auto es significativamente mayor, comparado con el costo de la bicicleta y de los patines juntos. El amigo entonces tiene una idea general de lo que piensas gastar en la compra y el costo de la bicicleta y los patines es ya irrelevante.

Así mismo los algoritmos. Para describir el comportamiento de un algoritmo se utilizan funciones como

$$n^3 + 12n + 500,$$

que para fines prácticos se puede decir n^3. Aquí la variable n representa el tamaño del problema a resolver, por ejemplo en un problema de ordenamiento, n podría ser la cantidad de números a ordenar, etc. La expresión matemática establece la relación entre el tamaño de la entrada y el tiempo de ejecución [medido en «momentos»].

Las razones de crecimiento son funciones que tienen un comportamiento específico con respecto al tamaño de la entrada n. Aquí una lista de algunas razones de crecimiento:

1. 2^{2^n}
2. $n!$
3. 4^n
4. 2^n
5. n^2
6. $n\texttt{log}(n)$
7. $\texttt{log}(n!)$
8. n
9. $2^{\texttt{log}(n)}$
10. $(\texttt{log}(n))^2$
11. $\sqrt{\texttt{log}(n)}$
12. $\texttt{log}(\texttt{log}(n))$
13. 1

Para tener una idea general, pero clara acerca del análisis de algoritmos, estudiemos este problema:

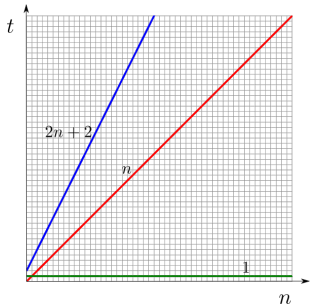
```
Problema::
Encontrar la suma de los primeros números enteros positivos.

Entrada::
n: número entero positivo

Salida::
El número entero que corresponde a la suma de los primeros enteros desde 1 hasta n inclusive.
```

```
1  def suma_enteros_1(n):
2      """
3      Obtiene la suma de los primeros n números enteros
4      n : número entero positivo
5      """
6      suma_total = 0
7      for i in range(1,n + 1):
8          suma_total = suma_total + i
9      return suma_total
```

El tiempo de ejecución de este algoritmo crece en forma proporcional a la cantidad de números que tiene que sumar, esto es que el algoritmo tardará mas a medida que se solicite un número más grande. Digamos que cada operación de suma toma 1 momento de tiempo, el momento de tiempo puede ser ajustado de acuerdo a las especificaciones de cada computadora, pero por ahora, y sólo para tener una idea general, todas las operaciones de este algoritmo toman la misma cantidad de tiempo, un momento.



Analicemos cada línea:

Línea	Instrucción	Num. Veces
6	`suma_total = 0`	1
7	`for i in range(1,n + 1):`	$(n+1)-(1)+(1) = n+1$
8	`suma_total = suma_total + i`	n
9	`return suma_total`	0
	Total:	$2n+2$

El tiempo de ejecución para un tamaño de entrada n lo escribimos $\texttt{T}(n)$, y en este caso $\texttt{T}(n) = 2n+2$ que es considerado en términos asintóticos [para evitar detalles] $\texttt{O}(n)$. Lo que indica que a medida que se solicite la suma de un número más grande, el tiempo de ejecución crecerá en proporción directa.

Este algoritmo tiene una razón de crecimiento que muchos algoritmos quisieran, es dcir un crecimiento lineal está bien, pero siempre podemos hacernos la pregunta **¿es posible hacerlo mejor?**. Esta es una pregunta clave, que exige un conocimiento profundo de técnicas matemáticas, estructuras de datos y en general del conocimiento involucrado en cada problema. Observa el siguiente algoritmo, que es la versión algorítmica de

$$\sum_{i=i}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

```
1  def suma_enteros_2(n):
2      """
3      Obtiene la suma de los primeros n números enteros
4      n : número entero positivo
5      """
6      suma_total = n * (n + 1) // 2
7      return suma_total
```

Analicemos cada línea:

Línea	Instrucción	Num. Veces
6	`suma_total = n * (n + 1) // 2`	1
7	`return suma_total`	0
	Total:	1

Este segundo algoritmo dice que sin importar el número n, que es pasado como parámetro, el resultado de la suma se obtendrá en el mismo tiempo, nunca se tardará mas o será más ágil. Por eso decimos que el tiempo de ejecución del algoritmo es **constante**, lo que escribimos $\mathtt{T}(n) = 1$.

2. Programación en Python

En este curso vamos a tomar `Python` como lenguaje base, el lenguaje en el que se deberán programar todos los ejercicios de programación. `Python` es un lenguaje relativamente nuevo, que data de finales de la década de los ochenta y principio de los noventa.

2.1. Razones para elegir `Python`

Hay al menos cinco razones por las cuales `Python` es una buena elección para estudiar algoritmos:

1. **Es simple y rápido**: La sintaxis del lenguaje se aprende rápido, hay relativamente pocas palabras clave que deben aprenderse. `Python` es razonablemente rápido, lo que es bueno, aunque en velocidad es peor que algunos otros lenguajes como `C` o `ensamblador`.

2. **Es débilmente tipificado**: Lo que significa que los tipos de dato se resuelven en tiempo de ejecución, por lo que en general no se debe uno preocupar demasiado por los tipos de dato, observa en el encabezado de la función `suma_entero_1`, donde se solicita un valor n, sin especificar qué tipo de dato debe ser.

3. **Propósito general**: Con un lenguaje de proósito general es posible escribir programas para cualquier proósito, desde los más pequeños programas hasta aquellos que se utilizan en Web o en laboratorios científicos. Algunas cosas es más difícil hacerlas que en otros lenguajes más especializados, pero así sucede con las herramientas multipropósito.

4. **Multiparadigma**: En un mismo programa se pueden utilizar diferentes paradigmas de programación, como programación orientada a objetos, programación imperativa o incluso es posible integrar la programación funcional.

5. **Comunidad de soporte**: Debido a que estge lenguaje es muy popular, hay una gran cantidad de **bibliotecas de funciones** disponibles para propósitos específicos, incluyendo soporte para diferentes motores de bases de datos, que lo hacen un lenguaje con mucho respaldo.

2.2. Instalación

Este curso utiliza como lenguaje de programación de apoyo a `Python` 3.6.9 pero cualquier versión a partir de la 3.0 es adecuada.

En sistemas operativos como `linux` o MacOS, python ya está instalado por defecto, pero quizá quieras instalar una nueva versión; en windows es posible que no esté instalado. Para saber si tienes disponible `Python` en tu sistema, puedes averiguarlo desde una terminal, escribiendo `python3 --version`, que en primer lugar invoca el programa `python3` y luego solicita la versión que se encuentra instalada.

```
$ python3 --version
Python 3.6.9
$ _
```

2.3. Instalación en Linux [Debian, Ubuntu, Mint...]

Si tienes la versión `Ubuntu 16.10` o más actual, fácilmente puedes intalar `Python` 3.6 con los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install python3.6
```

Si tienes otra versión of Ubuntu (p.ej.. la última versión LTS) o quieres utilizar una versión más actual de `Python`, te recomiendo utilizar otros repositorios para instalar `Python 3.8`:

```
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.8
```

2.4. Instalación en windows

Para instalar `Python` en windows debes seguir estos pasos:

1. Descarga `Python` en la última versión estable [3.7]. En https://www.python.org/downloads/ descarga el programa.

2. Inicia la instalacin ejecutando el archivo descargado ya sea `Python-3.7.0.exe` o `Python-3.7.0-amd64.exe` con doble clic. Si es necesario, confirma la ejecucin del programa como administrador.

3. En la ventana `Install Python 3.7.0 (64 bit)` activa las casillas: `Install launcher for all users (recommended)` y `Add Python 3.7 to PATH`. Después, selecciona la opción `Customize installation. Choose Location and features`.

4. En la ventana `Optional features` debes verificar que esten activadas todas las opciones y continúa la isntalación [`Next`].

5. En la ventana `Advanced Options` verifica que esten desactivadas las opciones `Download debugging symbols` y `Download debug binaries`, las demás sí deben estar marcadas. Escribe la ruta del directorio de instalación `C:\Python37` (o acepta la ruta por defecto) y comienza la instalación haciendo clic en el botón [`Install`].

6. Una vez que terminado el proceso de instalación, cierra [`Close`] la ventana y verifica la versión desde una terminal de comandos.

2.5. Entorno de desarrollo integrado [IDE]

Para hacer de la programación una tarea más rápida y productiva, se han hecho programas que integran un editor de texto y accesos a la revisión y ejecución del código dentro del mismo programa, este tipo de programas se conocen como `IDE`. Hay `IDE`s para exclusivamente para un lenguaje y hay `IDE`s que sirven para muchos lenguajes al mismo

tiempo. En este curso no preferimos uno sobre otro, puedes utilizar alguno que ya tengas instalado en tu computadora.

Para el uso de `Python` muchas personas recomiendan el entorno `Anaconda`, en su versión individual https://www.anaconda.com/products/individual.

Otras recomendaciones son `ATOM` https://atom.io/ o Visual Studio Code https://code.visualstudio.com/

2.6. Primeros pasos con Python

Esta sección te debe servir para conocer un poco el entorno de programación en `Python`, no es un curso de programación, sino apenas un esbozo general de cómo se programa en `Python`. Por favor si quieres un curso completo de programación en `Python`, hay una inmensa cantidad de tutoriales disponibles [y sin costo], que puedes descargar de Internet.

2.6.1. Variables

Las variables son símbolos a los que se les asocia con un valor que puede cambiar durante el transcurso del programa. Las variables en `Python` se escriben empezando con una letra y luego puedes escribir otras letras o números.

```
# Se define la variable a con el valor 10
a = 10;
# Se define la variable b con el valor 15.38
b = 15.38;
# Se define la variable c con el valor "hola mundo"
c = "hola mundo";
```

Al crear una nueva variable, no se declara su tipo de dato. El tipo de dato se resuelve en tiempo de ejecución.

```
>>> print(a,b,c)
10 15.38 hola mundo
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
>>>
```

2.6.2. Operadores

Estos son los operadores aritméticos que se utilizan en `Python`:

Operador	Símbolo	Ejemplo	Resultado
Suma	+	56 + 44	100
Resta o negativo	-	56 - 44 o -6	12 o -6
Multiplicación	*	64 * 2	128
División	/	45 / 4	11.25
Cociente	//	45 // 4	11
Residuo	%	45 % 4	1
Exponenciación	**	2**4	16

Los operadores booleanos producen un valor booleano, que puede ser `True` [cierto] o `False` [falso]. Los operadores booleanos son:

Operador	Símbolo	Ejemplo	Resultado
Conjunción	and	True and False	False
Disyunción	or	True or False	True
Negación	not	not False	True
Igualdad	==	10 == (5 * 2)	True
Diferencia	!=	15 != 23	True
Mayor que	>	10 > 100	False
Menor que	<	10 < 100	True

Tomemos por ejemplo este programa que se llama `comparaSumas.py`:

```
# Lee diferentes valores en linea y los guarda en un arreglo
A = input().split(" ")

# Convierte cada valor en un numero entero
B = list(map(int, V))

# Compara la suma de los primeros dos trminos con la suma de los otros dos
C = B[0] + B[1] == B[2] + B[3]

print(C)
```

Puedes probar el programa desde el REPL de `Python`, ejecutando el archivo `comparaSumas.py`:

```
>>> exec(open('comparaSumas.py').read())
1 5 3 3
True
>>>
```

Desde la terminal también se puede probar el programa.

```
$ python3 comparaSumas.py
1 5 3 3
True
$
```

2.6.3. Expresiones condicionales

Sentencias condicionales `if` Una de las instrucciones fundamentales de cualquier lenguaje de programación, son las expresiones condicionales de la forma `si-entonces`. En `Python` la instrucción `if` tiene varias formas de utilizarse, la más simple consiste de la palabra clave `if`, una expresión booleana y un signo `:` que indica elfinal de la condición, luego el código a ser ejecutado en caso de que la condición sea `True`. Observa el siguiente ejemplo:

```
1  # Ejemplo 1 del uso de if
2
3  x = 100;
4  if x > 0:
5    print("El valor de", x, "es mayor que 0.")
6    print("--------------------------------")
7  x = x + 5
```

Observa que la sentencia `if` abarca desde la línea 4 hasta la línea 6. La expresión booleana es `x > 0` y es seguida de `:` luego hay dos instrucciones [líneas 5 y 6] que se encuentran **en el mismo nivel de identación**, lo que significa que están dentro del mismo ámbito de definición.

```
1  # Ejemplo 2 del uso de if
2
3  x = -100;
4  if x > 0:
5    print("El valor de", x, "es mayor que 0.")
6    print("--------------------------------")
7  else:
8    print("El valor de", x, "NO es mayor que 0.")
9    print("--------------------------------")
10
11 x = x + 5
```

Aún hay otra forma de utilizar `if` y es cuando se tienen diferentes casos. considera la expresión matemática:

$$\texttt{abs}(x) = \begin{cases} \texttt{si } x < 0 & , -x \\ \texttt{si } x == 0 & , 0 \\ \texttt{eoc} & , x \end{cases}$$

Claro que la función valor-absoluto se puede hacer de otras maneras, pero el objetivo es ilustrar una lista de casos diferentes.

```
# Ejemplo 2 del uso de if

x = -100;
if x < 0:
  print("El valor absoluto de", x, "es ", -x)
  print("--------------------------------")
elif x == 0:
  print("El valor absoluto de", x, "es 0")
  print("--------------------------------")
else:
  print("El valor absoluto de", x, "es ", x)
  print("--------------------------------")

x = x + 5
```

2.6.4. Ciclos `for` y `while`

La sintaxis básica que se utiliza en las sentencias `for` es la siguiente:

```
for <id> in <secuencia>:
  <expresion>
  ...+
```

Por ejemplo, si queremos iterar una función sobre todos los elementos de una secuencia ⟨`'mexico', 'colombia', 'ecuador'`⟩, podemos crear el ciclo:

`ciudades.py`

```
# Ejemplo 1: ciclo for en python

ciudades = ['Cancn', 'Puerto Vallarta', 'Hermosillo']

for ciudad in ciudades:
  print(i, ciudad)
```

```
abcabc:~$ python3 ciudades.py
1  Cancn
2 Puerto Vallarta
3 Hermosillo
```

Observa que en el ciclo `for`, el identificador se instancia con cada uno de los elementos de la secuencia [también llamada iterador]. Hay algunos modificadores para trabajar con ciclod `for`, estudia los cambios hechos en el siguiente programa:

`ciudades.py`

```
# Ejemplo 2: ciclo for en python
ciudades = ['Cancn', 'Puerto Vallarta', 'Hermosillo']

print(list(enumerate(ciudades)))

for i, ciudad in enumerate(ciudades, start=1):
  print(i, ciudad)
```

Otro modo muy común de iterar es por medio de la instrucción `while`. A diferencia de `for`, en `while` se continúa el ciclo de instrucciones **mientras** una condición específica permanece siendo verdadera; cuando deje de ser verdadera, entonces se termina el ciclo.

`pares.py`

```
# Ejemplo 3: ciclo while en python

numeros_pares = 2;

while numeros_pares <= 10:
    print(numeros_pares);
    numeros_pares = numeros_pares + 2;
```

```
abcabc:~$ python3 pares.py
2
4
6
8
10
```

2.6.5. Funciones

Sin duda, las funciones son una de las maneras más sencillas y potentes de especificar tareas particulares. Todos los lenguajes de programación modernos, permiten encapsular algunas instrucciones que juntas realizan una **función** en particular.

En `Python` las funciones se declaran mediante la palabra clave `def`, escribiendolo así:

```
def <identificador>(<param> ...):
  <instr> ...
  return <valor>
```

Por ejemplo, queremos resolver el siguiente problema computacional.

Ejemplo 2: Valor absoluto

Se desea un programa en `Python` que reciba un número real, y devuelva el valor absoluto de ese número.

Entrada: Un número `n`.

Salida: Un número real con el valor absoluto del número `n`, de acuerdo a la siguiente regla:

$$\texttt{vabs}(n) = \begin{cases} \text{Si } n < 0 & , -n; \\ \text{Si } n = 0 & , 0; \\ \text{Si } n > 0 & , n; \end{cases}$$

Entrada	Salida
-56	56.0
0	0.0
23.9	23.9

vabsoluto.py

```
1  def vabs(n):
2      """
3      Devuelve el valor absoluto de un nmero n
4      n : numero
5      """
6      if (n<0):
7          return -n
8      elif (n==0):
9          return 0
10     else:
11         return n
12
13 v = float(input())
14 print(vabs(v))
```

Ejercicios

Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

- 3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.
- 2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.
- 1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.
- 0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Escribe un algoritmo que resuelva el siguiente problema.

```
PROBLEMA: Determinar el número que es mayor dentro de una colección de 4 números.
ENTRADA: Cuatro números enteros.
SALIDA: El número mayor de los cuatro de entrada.
```

Entrada	Salida
`3 6 9 1`	`9`
`1 3 3 3`	`3`
`4 5 -6 0`	`5`

2. Considera nuevamente el algoritmo hecho en el ejercicio 1. Escribe un razonamiento sobre las 5 características importantes de todo algoritmo [página 2]. ¿Tu algoritmo tiene estas características?¿Porqué?

3. Busca en Internet un ejemplo del tipo de algorimo mencionado en la página 4.

4. Dentro de las razones para utilizar `Python` se mencionó que es multiparadigma. En este ejercicio debes leer, comprender y explicar qué es y para qué sirve cada uno de los paradigmas mencionados [orientado a objetos, imperativo y funcional].

5. Para que tengas una clara idea de cómo se comportan las funciones que indican la razón de crecimiento, grafica cada una de las funciones enlistadas en la página 6. El eje horizontal debes etiquetarlo con n, que es el tamaño del problema a resolver. El eje vertical debe llevar la equiqueta $T(n)$ que es el tiempo de ejecución. Los límites de la gráfica son $0 \leq n \leq 100$ y $0 \leq T(n) \leq 100$. Puedes agrupar 3 o 4 en la misma gráfica.

Cuidado!

Los siguientes ejercicios son ejercicios de programación. Observa cuidadosamente las siguientes notas:

- Escribe correctamente el nombre del archivo, eso facilita la tarea de revisión.
- En los programas donde se solicita una entrada de algún tipo, debes solicitar la entrada **SIN** ningún texto previo; por ejemplo, si el programa requiere un número entero `n`, tu programa no debe solicitar el número con textos como `Dame un número:` , sino simplemente esperar hasta que el usuario escriba el número solicitado.
- Puedes confiar en que los tipos de datos de entrada se respetarán al hacer la evaluación del programa, esto es que si se solicita un número entero, siempre se probará el programa con números enteros. Los programas siempre recibirán el tipo de dato solicitado.
- Estas observaciones valen para todos los programas del curso.

6. Escribe un programa que calcule el producto de dos vectores de tamaño n. El producto del vector $\langle a_1 \dots a_n \rangle$ por el vector $\langle b_1 \dots b_n \rangle$ está dado por

$$\sum_{i=1}^{n} a_i * b_i$$

Tu programa se debe llamar `prodVec.py` y debe recibir como **entrada** un número n, que indica la longitud de los dos vectores; una lista de n números enteros, que son las entradas del primer vector; una lista de n números enteros, que son las entradas del segundo vector. La **salida** debe ser un único número que representa el producto de ambos vectores.

Entrada	Salida	Descripción
3 1 2 3 4 5 6	32	$(1 \times 4) + (2 \times 5) + (3 \times 6)$

*Problema tomado de OmegaUp https://omegaup.com/arena/problem/Producto-punto-de-dos-vectores

7. Escribe un programa que evalúe la siguiente fórmula con los valores de tres números dados como entrada:

$$\frac{(\frac{2x+y}{z})(y^3 - z)}{\frac{x+2y+3z}{z-2y-3x} + x^2 + z^2}$$

Tu programa se debe llamar `formulaGigante.py`, debe recibir como **entrada** tres números reales y debe ofrecer como **salida** un número real que corresponde a la evaluación de la expresión. Por ejemplo:

Entrada	Salida	Descripción
1.5 4 4.5	1.578947	Tu salida puede ser ligeramente diferente.

*Problema tomado de OmegaUp https://omegaup.com/arena/problem/Evaluando-una-formula-gigante/#problems/Evaluando-una-formula-gigante