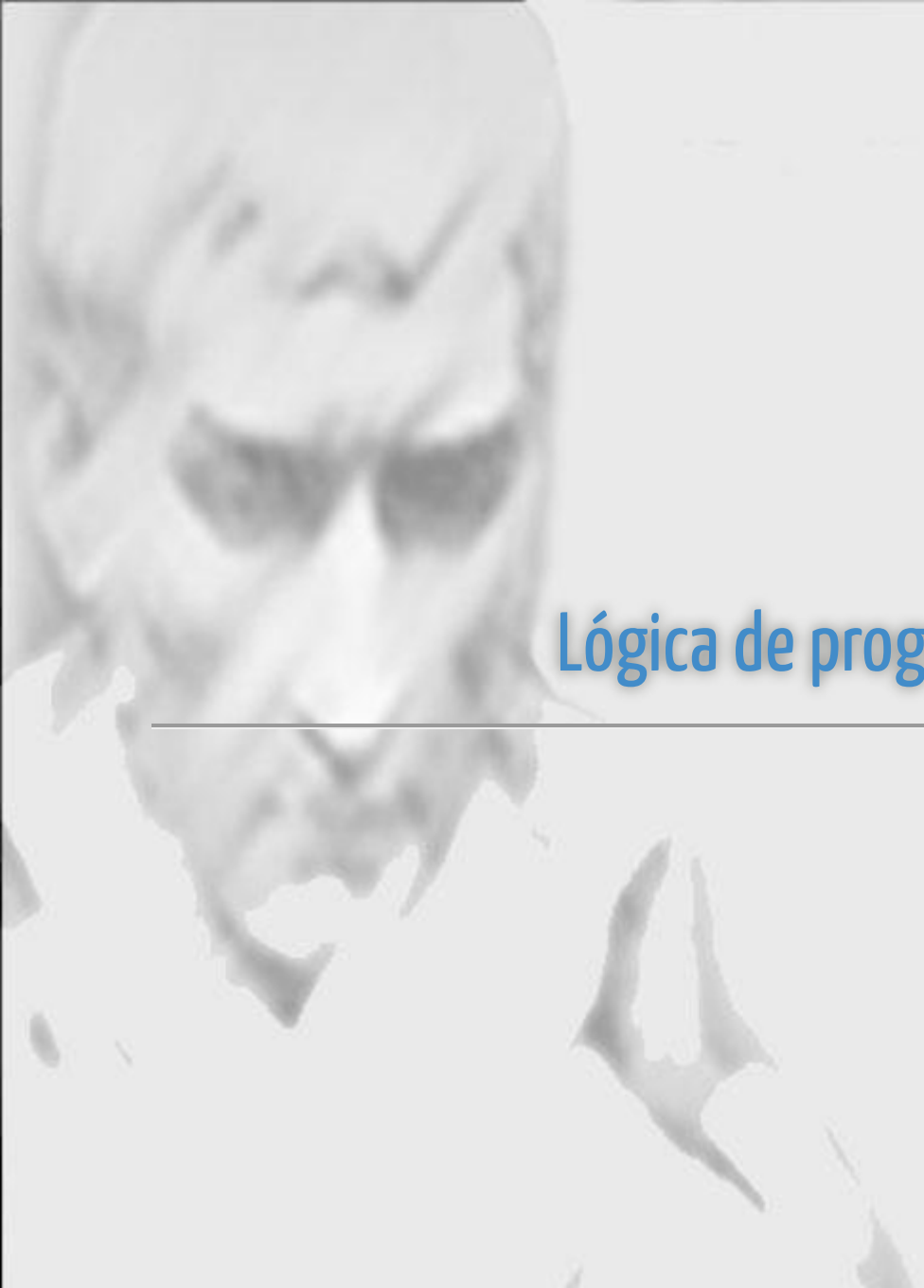


Lógica de programación

Lógica de programación

Contenidos

- Algoritmos y manejo de información
- Sentencias básicas
- Estructuras de control
- Tipos de datos compuestos
- Abstracción y definición de procedimientos y funciones
- Objetos y Clases
- Scripting
- Programación declarativa

Algoritmos y manejo de información

Introducción a los algoritmos:

El término **algoritmo** se puede definir como un procedimiento de resolución de problemas y de forma general se usa para el conjunto de ***reglas*** que una máquina sigue para lograr un ***objetivo*** particular.

Sin embargo, no siempre se aplica a la actividad mediada por computadora. El término puede usarse con la misma precisión de los pasos seguidos para hacer una pizza como para el análisis de datos por computadora.

El algoritmo a menudo se combina con palabras que especifican la actividad para la cual se ha diseñado un conjunto de reglas. Por ejemplo:

- Un algoritmo de búsqueda, por ejemplo, es un procedimiento que determina qué tipo de información se recupera de una gran masa de datos.
- Un algoritmo de encriptación es un conjunto de reglas mediante las cuales se codifica información o mensajes para que personas no autorizadas no puedan leerlos.

Algoritmos y manejo de información

Algorítmica:

La algorítmica es la ***ciencia*** que estudia a los ***algoritmos***, de los cuales estudia su ***eficiencia*** (la cantidad de recursos informáticos que requieren para completar su tarea) y su ***complejidad*** (cómo crece el tiempo de ejecución del algoritmo a medida que aumenta el tamaño de entrada).

Algoritmo:

De forma clasica un algoritmo es un ***procedimiento*** que una computadora o un ser humano sigue para resolver un problema. Pueden realizar cálculos, procesamiento de datos y tareas de razonamiento automatizado.

Un algoritmo informático se escribe en un lenguaje de programación que la computadora puede entender, pero para su desarrollo se usan representaciones intermedias que van desde representaciones informales como escritura en prosa, diagramas de flujo, pseudocódigo hasta llegar a su representación en un lenguaje de programación.

Algoritmos y manejo de información

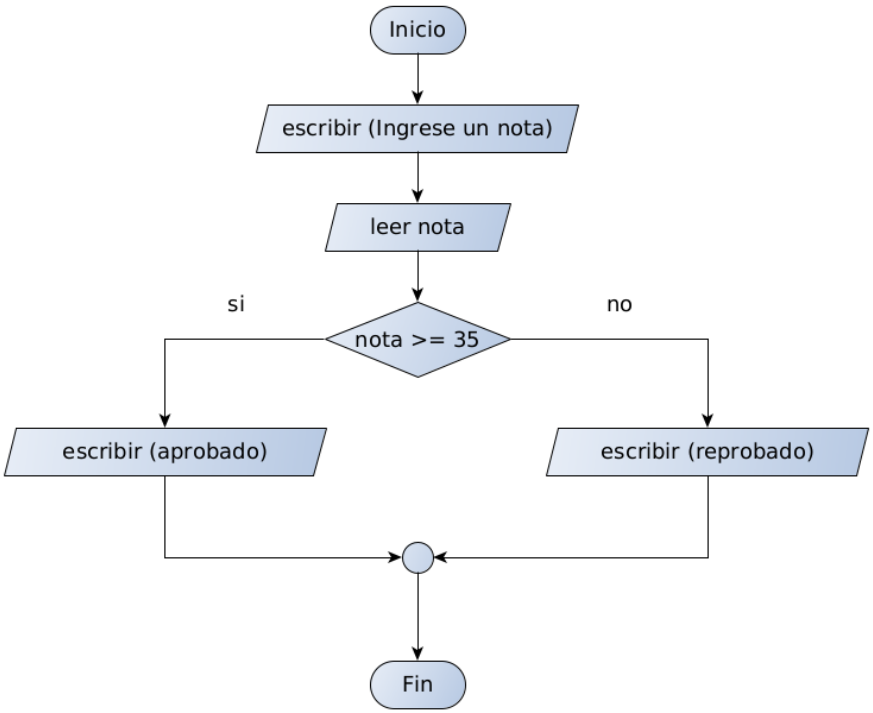
Características clave de los algoritmos son:

Hay ciertos requisitos que debe cumplir un algoritmo:

- **Definitividad**: cada paso del proceso se establece con precisión.
- **Computabilidad**: cada paso en el proceso puede ser realizado por una computadora.
- ***Finitud***: El programa eventualmente terminará exitosamente es decir el algoritmo se detiene después de un número finito de pasos.
- **Fácilidad de entender**: los algoritmos deben ayudar a comprender la solución a un problema.
- ***Corrección***: su comportamiento debe ser para el cual fué diseñado el algoritmo.
- ***Precisión***: los pasos se establecen (definen) con precisión.
- ***Generalidad***: el algoritmo se aplica a toda la distribución posible de entradas como se indica.

Algoritmos y manejo de información

Ejemplo diagrama de flujo:



Algoritmos y manejo de información

Ejemplo pseudocódigo:

```
procedimiento calificar:
inicio
    escribir ("ingrese una nota:")
    leer nota
    si nota >= 35 entonces:
        escribir ("aprobado")
    sino:
        escribir ("reprobado")
    finsi
fin
```

Ejemplo python:

```
def calificar():
    nota = input("ingrese una nota:")
    if nota >= 35:
        print("aprobado")
    else:
        print("reprobado")
```

Sentencias básicas

Datos

Los datos representan los componentes que representan la información que manipulan nuestros algoritmos y programas.

Tipos de datos:

El tipo de datos es una forma de clasificar los datos para su manipulación y esto determina tanto los valores que se pueden usar con el tipo de datos como el tipo de operaciones que se pueden realizar.

Hay dos tipos de datos:

- primitivos
- derivados

Sentencias básicas

Datos

tipos de datos primitivos

Los tipos de datos para los que un idioma tiene soporte incorporado se conocen como tipos de datos primitivos. Por ejemplo, la mayoría de los lenguajes de programación proporcionan soporte para los siguientes tipos de datos:

- numéricos (enteros y flotantes)
- booleanos (verdadero y falso)
- carácteres y cadenas de carácteres

Sentencias básicas

Datos

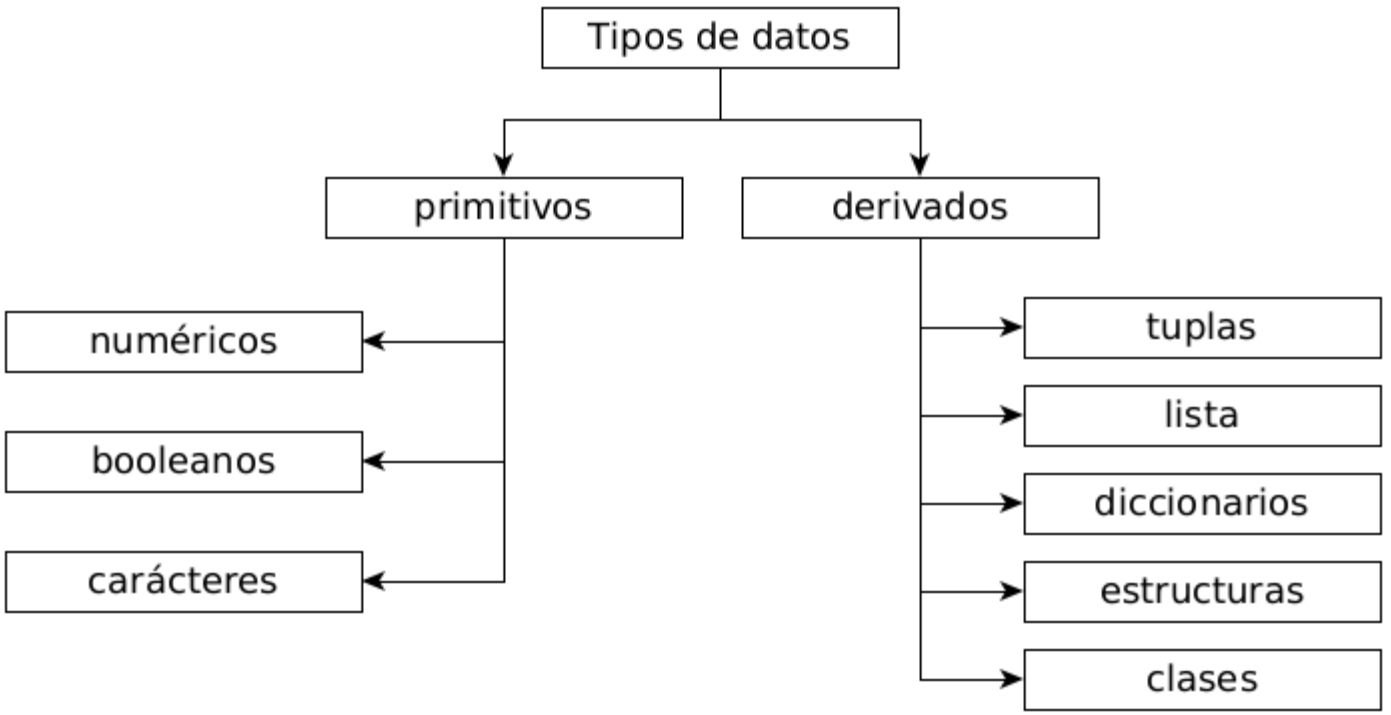
tipos de datos derivados

Estos tipos de datos normalmente se crean mediante la combinación de tipos de datos primitivos y las operaciones asociadas en ellos. Por ejemplo, muchos lenguajes de programación proporcionan soporte para los siguientes tipos de datos:

- tuplas
- listas
- colas
- estructuras
- diccionarios
- clases

Sentencias básicas

Datos



Sentencias básicas

Variables

Una variable es un nombre asignado a una entidad que puede adquirir un valor cualquiera dentro de un conjunto de valores. Es decir, una entidad cuyo valor puede cambiar a lo largo del programa. En un programa de computador se puede asumir que una variable es una posición de memoria donde los valores asignados pueden ser reemplazados o cambiados por otros valores durante la ejecución del programa.

```
numero_uno = 15
numero_dos = 2
flotante_uno = 2.5
flotante_dos = 3.0
variable = "texto"
var_verdadera = True
```

Sentencias básicas

Constante

Nombre asignado a una entidad al cual se asigna un valor que mantiene sin cambios durante el programa.

```
PORT_DB_SERVER = 3307
USER_DB_SERVER = "root"
PASSWORD_DB_SERVER = "123456"
DB_NAME = "nomina"
```

Reglas y convención de nombres para las variables y constantes:

- Nunca use símbolos especiales como !, @, #, $, %, etc.
- El primer carácter no puede ser un número o dígito.
- Los nombres de constantes y variables deberían tener la combinación de letras en minúsculas o MAYÚSCULAS o dígitos o un underscore (_).
- No pueden usarse como identificadores, las palabras reservadas .

Sentencias básicas

Operaciones

Operación: Acción por medio de la cual se obtiene un resultado de un operando. Ejemplos: sumar, dividir, unir, restar.

Operando: número, texto, valor lógico, variable o constante sobre la cual es ejecutada una operación.

Operador: símbolo que indica la operación que se ha de efectuar con el operando, por ejemplo, + / - * > == ≠ ≥ =

```
suma = numero_uno + numero_dos
resta = numero_uno - numero_dos
producto = numero_uno * numero_dos
division_uno = numero_uno / numero_dos
division_dos = numero_uno // numero_dos
division_tres = flotante_uno / flotante_dos
potencia = numero_uno ** numero_dos
repeticion = variable * numero_dos
concatenado = variable + " - " + str(numero_dos)
```

Sentencias básicas

Operadores Aritméticos

Entre los operadores aritméticos, podemos encontrar los siguientes:

Símbolo	Significado	Ejemplo	Resultado
+	Suma	a = 10 + 5	a es 15
-	Resta	a = 12 - 7	a es 5
-	Negación	a = -5	a es -5
*	Multiplicación	a = 7 * 5	a es 35
**	Exponente	a = 2 ** 3	a es 8
/	División	a = 12.5 / 2	a es 6.25
//	División entera	a = 12.5 / 2	a es 6.0
%	Módulo	a = 27 % 4	a es 3

Sentencias básicas

Operadores lógicos

Entre los operadores lógicos, podemos encontrar los siguientes:

Símbolo	Significado	Ejemplo	Resultado
and	y lógico	True and False	False
or	o lógico	True or False	True
not	negación	not True	False

Sentencias básicas

Operadores comparación

Entre los operadores de comparación, podemos encontrar los siguientes:

Símbolo	Significado	Ejemplo	Resultado
==	igual	10 == 5	False
<	menor	2 < 7	True
>	mayor	1 > -5	True
<=	menor o igual	7 <= 5	False
>=	mayor o igual	2 >= 2	True
!=	diferente	4 != 1	True

Sentencias básicas

Expresiones

Son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones o acciones. Cada expresión toma un valor que se determina evaluando los valores de sus variables, constantes y operadores.

```
nombre = input("ingrese su nombre:")
cadena = "Hola " + nombre
print(cadena)

numero_uno = int(input("ingrese un número:"))
for i in range(4):
  print(numero_uno + 1)
```

Sentencias básicas

Prioridad de Operadores

- Indica en qué orden debe aplicarse diferentes operaciones sobre un conjunto de valores. Permiten aplicar los operadores en el orden correcto.
- En caso de haber operadores del mismo nivel en una expresión, se evalúan en orden de aparición de izquierda a derecha.

operador	significado
()	paréntesis
^	potencia
*, /	multiplicación, division
+, -	suma, resta
==, !=, >, <, >=, <=	comparación
not	negación
and	y lógico
or	o lógico

Estructuras de control

Una estructura de control, es un bloque de código que permite agrupar instrucciones de manera controlada.

Dos tipos de estructuras de control:

- Estructuras de control condicionales
- Estructuras de control iterativas

Una estructura de control, entonces, se define de la siguiente forma:

```
inicio de la estructura de control:
  expresiones
```

o

```
inicio de la estructura de control {
  expresiones
}
```

Estructuras de control: condicionales

Las estructuras de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar.

La evaluación de condiciones, solo puede arrojar 1 de 2 resultados: verdadero o falso (True o False).

Para describir la evaluación a realizar sobre una condición, se utilizan operadores relacionales (o de comparación):

Símbolo	Significado	Ejemplo	Resultado
==	Igual que	5 == 7	Falso
!=	Distinto que	rojo != verde	Verdadero
<	Menor que	8 < 12	Verdadero
>	Mayor que	12 > 7	Falso
<=	Menor o igual que	12 <= 12	Verdadero
>=	Mayor o igual que	4 >= 5	Falso

Estructuras de control: condicionales

Y para evaluar más de una condición simultáneamente, se utilizan operadores lógicos:

Operador	Ejemplo	Resultado
and (y)	5 == 7 and 7 < 12	0 y 0 Falso
	9 < 12 and 12 > 7	1 y 1 Verdadero
	9 < 12 and 12 > 15	1 y 0 Falso
or (o)	12 == 12 or 15 < 7	1 o 0 Verdadero
	7 > 5 or 9 < 12	1 o 1 Verdadero
xor (o excluyente)	4 == 4 xor 9 > 3	1 o 1 Falso
	4 == 4 xor 9 < 3	1 o 0 Verdadero

Estructuras de control: condicionales

Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje: ***if*** (si), ***elif*** (sino, si) y ***else*** (sino).

Veamos algunos ejemplos:

```
if semaforo == verde:
     print("Cruzar la calle")
else:
     print("Esperar")

if compra <= 100:
     print("Pago en efectivo")
elif compra > 100 and compra < 300:
     print("Pago con tarjeta de débito")
else:
     print("Pago con tarjeta de crédito")

importe_a_pagar = total_compra
if total_compra > 100:
     tasa_descuento = 10
     importe_descuento = total_compra * tasa_descuento / 100
     importe_a_pagar = total_compra - importe_descuento
```

Estructuras de control iterativas

A diferencia de las estructuras de control condicionales, las iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.

Se dispone generalmente de dos estructuras de control cíclicas:

- El bucle while
- El bucle for

Bucle while

Este bucle, se encarga de ejecutar una misma acción “mientras que” una determinada condición se cumpla:

```
# -*- coding: utf-8 -*
anio = 2001
while anio <= 2012:
     print("Informes del Año", str(anio))
     anio += 1
```

Estructuras de control iterativas

Bucle while

La iteración anterior, generará la siguiente salida:

```
Informes del año 2001
Informes del año 2002
Informes del año 2003
Informes del año 2004
Informes del año 2005
Informes del año 2006
Informes del año 2007
Informes del año 2008
Informes del año 2009
Informes del año 2010
Informes del año 2011
Informes del año 2012
```

Podremos utilizar una estructura de control condicional, anidada dentro del bucle, y frenar la ejecución cuando el condicional deje de cumplirse, con la palabra clave reservada break:

```
while True:
    nombre = input("Indique su nombre: ")
    if nombre:
        break
```

Estructuras de control iterativas

Bucle for

El bucle for, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla:

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
for nombre in mi_lista:
     print(nombre)

mi_tupla = ('rosa', 'verde', 'celeste', 'amarillo')
for color in mi_tupla:
     print(color)
```

Otra forma de iterar con el bucle for, puede emular a while:

```
for anio in range(2001, 2013):
     print("Informes del Año", str(anio))
```

Tipos de datos compuestos

Tipos de datos complejos

Python, posee además de los tipos ya vistos, 3 tipos complejos o compuestos, que admiten una colección de datos.

Estos tipos son:

- Tuplas
- Listas
- Diccionarios

Estos tres tipos, pueden almacenar colecciones de datos de diversos tipos y se diferencian por su sintaxis y por la forma en la cual los datos pueden ser manipulados.

Tuplas

Una tupla es una variable que permite almacenar varios datos inmutables (no pueden ser modificados una vez creados) de tipos diferentes:

```
mi_tupla = ('cadena de texto', 15, 2.8, 'otro dato', 25)
```

Tipos de datos compuestos

Tuplas

Se puede acceder a cada uno de los datos mediante su índice correspondiente, siendo 0 (cero), el índice del primer elemento:

```
print(mi_tupla[1])  # Salida: 15
```

También se puede acceder a una porción de la tupla, indicando (opcionalmente) desde el índice de inicio hasta el índice de fin:

```
print(mi_tupla[1:4])  # Devuelve: (15, 2.8, 'otro dato')
print(mi_tupla[3:])   # Devuelve: ('otro dato', 25)
print(mi_tupla[:2])   # Devuelve: ('cadena de texto', 15)
```

Otra forma de acceder a la tupla de forma inversa (de atrás hacia adelante), es colocando un índice negativo:

```
print(mi_tupla[-1])  # Salida: 25
print(mi_tupla[-2])  # Salida: otro dato
```

Tipos de datos compuestos

Listas

Una lista es similar a una tupla con la diferencia fundamental de que permite modificar los datos una vez creados

```
mi_lista = ['cadena de texto', 15, 2.8, 'otro dato', 25]
```

A las listas se accede igual que a las tuplas, por su número de índice:

```
print(mi_lista[1]) # Salida: 15
print(mi_lista[1:4])  # Devuelve: [15, 2.8, 'otro dato']
print(mi_lista[-2])   # Salida: otro dato
```

Las lista NO son inmutables: permiten modificar los datos una vez creados:

```
mi_lista[2] = 3.8  # el tercer elemento ahora es 3.8
```

Las listas, a diferencia de las tuplas, permiten agregar nuevos valores:

```
mi_lista.append('Nuevo Dato')
```

Tipos de datos compuestos

Diccionarios

Mientras que a las listas y tuplas se accede solo y únicamente por un número de índice, los diccionarios permiten utilizar una clave para declarar y acceder a un valor:

```
mi_diccionario = {'clave_1': valor_1, 'clave_2': valor_2, 'clave_7': valor_7}
print(mi_diccionario['clave_2'])  # Salida: valor_2
```

Un diccionario permite eliminar cualquier entrada:

```
del(mi_diccionario['clave_2'])
```

Al igual que las listas, el diccionario permite modificar los valores

```
mi_diccionario['clave_1'] = 'Nuevo Valor'
```

Abstracción y definición de procedimientos y funciones

Definiendo funciones

En Python, la definición de funciones se realiza mediante la instrucción def más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre.

Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone, irá identado con 4 espacios:

```
def mi_funcion():     # aquí el algoritmo
```

Una función, no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
     print "Hola Mundo"

funcion()
```

Abstracción y definición de procedimientos y funciones

Definiendo funciones

Cuando una función, haga un retorno de datos, éstos, pueden ser asignados a una variable:

```
def funcion():
     return "Hola Mundo"

frase = funcion()
print(frase)
```

Abstracción y definición de procedimientos y funciones

Parámetros de funciones

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
def mi_funcion(nombre, apellido):     # algoritmo
```

Los parámetros, se indican entre los paréntesis, a modo de variables, a fin de poder utilizarlos como tales, dentro de la misma función.

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de variables de ámbito local. Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre, apellido
    print(nombre_completo)
```

Si quisiéramos acceder a esas variables locales, fuera de la función, obtendríamos un error.

Abstracción y definición de procedimientos y funciones

Parámetros por omisión

En Python, también es posible, asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def saludar(nombre, mensaje='Hola'):
     print(mensaje, nombre )

saludar('Pepe Grillo')  # Imprime: Hola Pepe Grillo
```

Keywords como parámetros

En Python, también es posible llamar a una función, pasándole los argumentos esperados, como pares de *clave=valor*:

```
def saludar(nombre, mensaje='Hola'):
     print(mensaje, nombre)

saludar(mensaje="Buen día", nombre="Juancho")
```

Abstracción y definición de procedimientos y funciones

Parámetros arbitrarios

Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de tupla. Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco (*):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios):
     print parametro_fijo
     # Los parámetros arbitrarios se corren como tuplas
     for argumento in arbitrarios:
              print argumento

recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1', 'arbitrario 2', 'arbitrar
```

Abstracción y definición de procedimientos y funciones

Parámetros arbitrarios

Es posible también, obtener parámetros arbitrarios como pares de *clave=valor*. En estos casos, al nombre del parámetro deben precederlo dos astericos (**):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios, **kwords):
     print parametro_fijo
     for argumento in arbitrarios:
              print argumento
     # Los argumentos arbitrarios tipo clave, se recorren como los diccionarios
     for clave in kwords:
              print "El valor de", clave, "es", kwords[clave]

recorrer_parametros_arbitrarios("Fixed", "arbitrario 1", "arbitrario 2", "arbitrar
                                clave1="valor uno", clave2="valor dos")
```

Abstracción y definición de procedimientos y funciones

Desempaquetado de parámetros

Puede ocurrir además, una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla o un diccionario:

- En el caso de que vengan en una lista o tupla, el signo asterisco (*) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función.
- El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se encuentren disponibles en un diccionario. Aquí, deberán pasarse a la función, precedidos de dos asteriscos (**).

```
def calcular(importe, descuento):
     return importe - (importe * descuento / 100)

datos = [1500, 10]
print(calcular(*datos))

datos = {"descuento": 10, "importe": 1500}
print(calcular(**datos))
```

Abstracción y definición de procedimientos y funciones

Llamadas recursivas

Se denomina llamada recursiva (o recursividad), a aquellas funciones que en su algoritmo, hacen referencia sí misma. Las llamadas recursivas suelen ser muy útiles en casos muy puntuales, pero debido a su gran factibilidad de caer en iteraciones infinitas, deben extremarse las medidas preventivas adecuadas.

Python admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma, de igual forma que lo hace cuando llama a otra función.

```
def jugar(intento=1):
     respuesta = input("¿De qué color es una naranja? ")
     if respuesta != "naranja":
          if intento < 3:
               print ("\nFallaste! Inténtalo de nuevo")
               intento += 1
               jugar(intento)  # Llamada recursiva
          else:
               print("\nPerdiste!")
     else:
          print("\nGanaste!")

jugar()
```

Objetos y Clases

Un sistema se califica como OrientadoaObjetos cuando reúne las características de: abstracción, encapsulación, herencia y polimorfismo; y los conceptos básicos que las forman: objetos, mensajes, clases, instancias y métodos.

Conceptos Básicos:

- ***Un objeto*** es una encapsulación abstracta de información, junto con los métodos o procedimientos para manipularla. Un objeto contiene operaciones que definen su comportamiento y variables que definen su estado entre las llamadas a las operaciones.
- ***Una clase*** equivale a la generalización o abstracción de un tipo específico de objetos.
- ***Un mensaje*** representa una acción a tomar por un determinado objeto.
- ***Una instancia*** es la concrección de una clase.
- ***Un método*** consiste en la implementación en una clase de un protocolo de respuesta a los mensajes dirigidos a los objetos de la misma.

Objetos y Clases

Características:

La abstracción: Consiste en la generalización conceptual de un determinado conjunto de objetos y de sus atributos y propiedades, dejando en un segundo término los detalles concretos de cada objeto.

La encapsulación: Se refiere a la capacidad de agrupar y condensar en un entorno con límites biendefinidos distintos elementos. La Clase es una encapsulación porque constituye una cápsula o saco que encierra y amalgama de forma clara tanto los datos de que constan los objetos como los procedimientos que permiten manipularlos.

La herencia: Se aplica sobre las clases. O sea, de alguna forma las clases pueden tener descendencia, y ésta heredará algunas características de las clases "padres". Si disponemos las clases con un formato de árbol genealógico, tenderemos lo que se denomina una estructura jerarquizada de clases.

El polimorfismo: como su nombre lo indica sugiere múltiples formas, se refiere a la posibilidad de acceder a un variado rango de funciones distintas a través del mismo interfaz. O sea, un mismo identificador puede tener distintas formas.

Objetos y Clases

Otros Conceptos

- ***Agregación***: Composición de un objeto por otros. Es una relación más débil que la que existe entre el atributo y el objeto al cual pertenece, y más fuerte que una asociación.
- ***Concurrencia***: Propiedad que distingue un objeto activo de otro inactivo.
- ***Persistencia***: Es la propiedad de un objeto cuya existencia trasciende el tiempo y/o el espacio (ej. el objeto continua existiendo luego de que su creador deja de existir / la ubicación de un objeto se mueve a un espacio de direcciones diferente de aquella donde fue creada).
- ***Visibilidad***: capacidad de restringir el acceso a atributos y servicios de un objeto. Particularmente importante en el diseño e implementación. (ej.: público / protegido / privado)

Objetos y Clases

El mecanismo de clases de Python agrega clases al lenguaje con un mínimo de nuevas sintaxis y semánticas.

Las clases en Python mantienen el poder completo de las características más importantes de las clases:

- el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobreescribir cualquier método de su(s) clase(s) base
- un método puede llamar al método de la clase base con el mismo nombre.

Los objetos pueden tener una cantidad arbitraria de datos. todos los miembros de las clases (incluyendo los miembros de datos), son públicos, y todas las funciones miembro son virtuales.

No hay atajos para hacer referencia a los miembros del objeto desde sus métodos: la función método se declara con un primer argumento explícito que representa al objeto, el cual se provee implícitamente por la llamada.

Objetos y Clases

Las clases mismas son objetos. Esto provee una semántica para importar y renombrar.

Los tipos de datos integrados pueden usarse como clases base para que el usuario los extienda.

También, la mayoría de los operadores integrados con sintaxis especial (operadores aritméticos, de subíndice, etc.) pueden ser redefinidos por instancias de la clase.

Objetos y Clases

Ejemplo de POO

```
class UnidadDeTiempo:
      def __init__(self):
            valor = 0
            tope = 59

      def avanzar(self):
            if self.valor < self.tope:
                  self.valor = self.valor+1
            else:
                  self.valor = 0;

      def resetear(self):
            self.valor = 0

      def getValor(self):
            return self.valor
```

Objetos y Clases

Ejemplo de POO

```
class Hora(UnidadDeTiempo):
      def __init__(self):
            self.valor=0
            self.tope=23

class Minuto(UnidadDeTiempo):
      def __init__(self):
            self.valor=0
            self.tope=59
```

Ejemplo de POO

```
class Cronometro:
      def __init__(self):
            self.h = Hora()
            self.m = Minuto()

      def avanzar(self):
            self.m.avanzar()
              if(self.m.getValor()==0):
                    self.h.avanzar()

      def getTiempo(self):
            return "{02d}:{02d}".format(self.h.getValor(), self.m.getValor())
```

Scripting

El término “script” se refiere a un guión y es un concepto tomado de las artes escénicas, en estas las instrucciones son interpretadas (ejecutadas) por los actores (programas) en un orden establecido.

Un lenguaje de scripting es un lenguaje de programación que tiene soporte a la escritura de guiones (scripts) y son usados en la automatización de tareas.

A diferencia de un programa tradicional un programa de scripting permite resolver en pocas líneas grandes tareas gracias a su alto nivel de abstracción en el cual enlazan componentes que internamente realizan dichas tareas.

Características de Scripting

- Lenguajes que dan soporte a la escritura de guiones (scripts).
- Son de un dominio específico (run-time environment).
- Son interpretados (no requieren pre-compilación).
- Sirven para automatizar tareas
- Son lenguajes de muy alto nivel.

Scripting

Características de Scripting

- Algunos ejemplos de lenguajes de scripting son: bash, JavaScript, VBA (Visual Basic para aplicaciones), PERL, TCL y Python.
- Son diseñados con una sintaxis y semántica simple.
- Son usados por usuarios y programadores finales para producir secciones de programas.
- Usualmente son creados por los mismos usuarios que los ejecutan (automatización de tareas).
- Creados frecuentemente para tareas interactivas mediante una consola de comandos.
- Se basan en la existencia de componentes creados en otros lenguajes.
- Son generalmente de tipo typeless.

Scripting

Ejemplos de Scripting

```
#!/bin/sh
echo "Determina el mayor de los numeros enteros"
echo "Digita el primer numero"
read var1
echo "Digita el segundo numero"
read var2
if [ $var1 -gt $var2 ]
then
echo "El mayor es $var1"
fi
if [ $var2 -gt $var1 ]
then
echo "El mayor es $var2"
fi
if [ $var1 -eq $var2 ]
then
echo "Son iguales"
fi
```

Scripting

Ejemplos de Scripting

```
#!/usr/bin/python

# cambiar los nombres de los archivos a mayúsculas

from os import path, listdir, rename

lista = listdir('archivos')

for a in lista:
    if path.exists('archivos/' + str(a)):
        rename('archivos/' + str(a),
               'archivos/' + str(a).upper())
    else:
        print("el archivo %s no existe" % str(a))
```

Programación declarativa

La programación declarativa es un estilo de programación en el que el programador especifica ***qué debe computarse*** en lugar de especificar ***cómo deben realizarse los cómputos***.

"programa = lógica + control" (Kowalski)

"algoritmos + estructuras de datos = programas" (Wirth)

La programación declarativa es un término que agrupa los siguientes paradigmas de programación:

- Programación lógica. Los problemas se representan por medio de lógica matemática.
- Programación funcional. Todo se resuelve por medio de la evaluación de funciones matemáticas.
- Lenguajes de dominio específico (DSLs). Lenguajes descriptivos para un propósito específico, tales como SQL.
- Lenguajes híbridos. Un ejemplo son los archivos "make" que combinan la descripción de dependencias entre componentes, con instrucciones imperativas para compilar o instalar una aplicación.

Programación declarativa

Un ejemplo

Un algoritmo imperativo:

```
suma = 0;
for i in range(100):
  suma += i
```

Ahora escrito de forma declarativa:

```
suma = Sum(1, 100)
```

"deberíamos estar haciendo las cosas en forma declarativa (···) no deberíamos estar escribiendo tanto código procedural". (Bill Gates)

Referencias Bibliográficas