



CLASE N°8

Python reloaded



- Un lenguaje de programación:
 - Interpretado
 - De **propósito general** que sustenta diferentes paradigmas de programación
 - Lo usamos para **programación imperativa**
- Podemos interactuar con el intérprete

El **prompt** indica que
el intérprete está listo
para recibir
instrucciones

The screenshot shows the Python Shell window of the IDLE 2.6.6 application. The title bar reads "74 Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the Python version information: "Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32) [MSC v.1500 32 bit (Intel)] on win32". Below this, a warning message about the connection to the subprocess via a loopback interface is shown. At the bottom of the window, the Python prompt ">>>" is visible, indicating that the interpreter is ready for input. A red arrow points from the explanatory text above to this prompt area.

Valores numéricos



- Hasta ahora hemos visto que Python maneja **tres tipos de datos numéricos**:

- Enteros (*int*): 2, -2, -205, 1024

[$-2147483648, 2147483647]$

- Enteros largos (*long*): 2L, -2L, -205L, 1024L, 2147483648L

- Flotantes de 64 bits (*float*): 2.1, -2.56, -3.234e-2

Valor más pequeño: $2.2250738585072014 \times 10^{-308}$

Valor más grande: $1.7976931348623157 \times 10^{308}$

Menor diferencia entre dos números: $2.2204460492503131 \times 10^{-16}$

3

Expresiones aritméticas



- Python permite operar valores numéricos

- Operadores tradicionales con reglas de precedencia tradicional:

- **** ; + y -** unarios; *****, **/**, **%**; **+ y -**

- Las reglas de precedencia pueden cambiarse usando **paréntesis**

- Todos los operadores tienen **asociatividad** por la izquierda, **excepto** el operador potencia

- $1 + 2 + 3 = ((1 + 2) + 3)$
 - $2 ** 3 ** 4 = (2 ** (3 ** 4))$

4

Expresiones aritméticas



- Python prefiere **mantener el tipo de dato**
 - Vimos, por ejemplo, que si se operan **dos enteros**, entonces Python intentará devolver un resultado **entero**
 - Pero si se **combinan** tipos numéricos en una expresión aritmética, Python **generaliza** todos los valores antes de operarlos
 - Enteros largos son más generales que los enteros
 - Los flotantes son más generales que los enteros largos y los enteros

5

Expresiones aritméticas



- Python también cambia el **tipo de dato**
 - Cuando el resultado de una operación hace imposible mantener el tipo de dato original

```
>>> 2 ** 31
2147483648L
>>>
```

- Cuando cambiamos **explícitamente** el tipo de dato de un valor por medio de **funciones nativas**
 - **int(<expresión>)**, **long(<expresión>)**
 - **float(<expresión>)**

6

Funciones nativas



- Recordemos que las funciones nativas **vienen** con Python
 - Simplemente las **invocamos** escribiendo su nombre e indicando los **parámetros actuales** que necesitan
 - También hemos visto:

round(x, n) Redondea el valor flotante x a n decimales;
devuelve un flotante

abs(x) Devuelve el valor absoluto de x

pow(x, y) Calcula x elevado a y

7

Entrada



- Pero quizás la función nativa más importante que hemos visto sea **input()**
 - Toma como argumento un **mensaje** para el usuario
 - **Detiene la ejecución** del programa hasta que el usuario “entra” una expresión Python (usualmente por teclado)
 - **Devuelve el valor resultante** de evaluar la expresión ingresada
- Nos permite obtener **datos de entrada** para un programa

8

Memoria



- Los datos de entrada no serían de mucha utilidad si no se pudieran **recordar**
 - Podemos solicitar a Python que recuerde un valor **asignándolo a un nombre**
 - La sintaxis de una asignación es la siguiente:
`<identificador> = <expresión>`
 - Por ejemplo:
`valorInicial = input("¿Cuál fue el valor inicial?: ")`
 - Este nombre asociado a un valor se conoce como **variable**

9

Funciones importadas



- También hemos visto **extensiones** al lenguaje
 - Podemos **importar** funciones al lenguaje desde un **módulo**
`from <módulo> import <nombre función>`
 - Por ejemplo el módulo **math** define funciones matemáticas
`sin(), cos(), tan(), exp(), floor(), ceil(), etc.`
 - Algunos módulos **definen constantes**
`>>> from math import pi, e
>>> pi
3.1415926535897931
>>> e
2.7182818284590451
>>>`



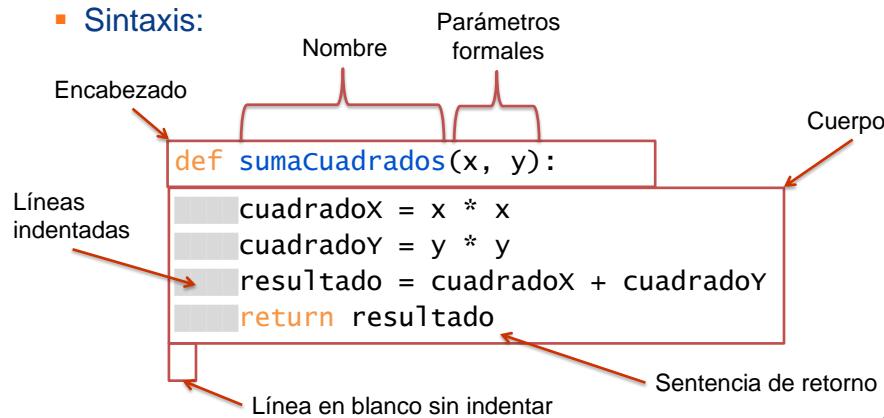
10



Funciones propias

- También podemos extender Python **creando** nuestras propias funciones

- Sintaxis:



Funciones propias



- El **cuerpo** de una función sigue reglas de **alcance**
 - Puede usar **variables globales** definidas fuera de la función
 - Define **variables locales** creadas dentro de la función
- Pero es importante mantener una buena **encapsulación**
 - Hacen una y sólo **una cosa** en forma completa
 - Siempre **devuelven un valor**
 - Se **comunican** con el exterior sólo a través de sus **argumentos**

12

Funciones propias



- Definir una función la deja **lista** para ser usada
- Para usarla, debemos **invocarla**
 - Aparece su **nombre** en **alguna expresión**
 - Se especifican los **parámetros actuales**, que se asocian **posicionalmente** a sus parámetros formales
 - Los parámetros actuales pueden ser **expresiones complejas**, que pueden incluir variables

13

Buenas prácticas de programación



- A medida que los programas se alargan, se hacen **más difíciles de entender** por seres humanos
- Es importante preocuparse de la **legibilidad** de los programas
- Vimos algunas **buenas prácticas** para ayudarnos en este aspecto
- Recordemos las más importantes

14

Buenas prácticas de programación



- En general:

- Estructurar el programa en **secciones de código**
 - Encabezado con una descripción general
 - Definición e importación de constantes
 - Definición e importación de funciones
 - Bloque principal del programa
- Usar **comentarios** para marcar claramente cada sección y describir el programa y cada función definida

15

Buenas prácticas de programación



- En el bloque principal:

- Se ordena la lógica que permite **resolver un problema**
- También conviene estructurarla en **secciones**:
 - Entrada de datos
 - Procesamiento, generalmente invocando funciones
 - Salida de datos (respuestas)
- También conviene **comentar** la lógica de la solución

16

Buenas prácticas de programación



- Usar **buenos identificadores** nos ahorra explicaciones
 - Nombres de variables usan **sustantivos**, escritos en **minúsculas** y capitalizando cuando el nombre es compuesto
 - Nombres de constantes usan **sustantivos**, escritos en **mayúsculas** y componiendo con guiones bajos
 - Nombres de funciones parten con un **verbo** y también se escriben en minúsculas y capitalizando
 - Los nombres escogidos deben ser indicativos (en el mundo real) de lo que se almacena o realiza

`sueldoAyudante, TASA_DE_INTERES, calculaDescuento()` 17

Valores y expresiones booleanas



- Python también maneja el tipo de dato **booleano**

- **Dos** posibles valores: `true` y `false`
 - Se obtienen **comparando** expresiones

```
>>> 5 * 5 <= 5 ** 2
True
>>>
```

- Los **operadores de comparación** disponibles son:

`==, !=, >, <, >=, <=`

- Todos tienen **igual precedencia**, la que es **menor** a la de los operadores aritméticos

18

Operadores booleanos



- Python permite componer expresiones booleanas con **operadores booleanos**

- Negación (**not**), conjunción (**and**) y disyunción (**or**)

```
>>> >>> 5 * 5 <= 5 ** 2 and not 10 < 5 or 3 == 3.0
True
>>>
```

- La **negación** tiene **menor precedencia** que los **operadores de comparación**, pero precede a la **conjunción**, la que a su vez precede a la **disyunción**

19

Decisión simple



- Las expresiones booleanas permiten **tomar decisiones** en un programa
- Vimos la estructura de **decisión simple**
 - Usa la sentencia **if**
 - Condiciona la **ejecución de un bloque** de sentencias a el cumplimiento de una **condición**
 - La condición es una **expresión booleana** y se cumple si su evaluación resulta en valor **true**

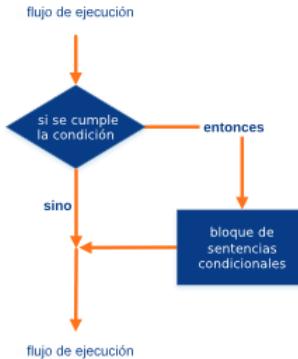
20



Decisión simple

- Sintaxis:

```
if <condición>:  
    <Bloque de sentencias condicionales>
```



21



Decisión alternativa

- También vimos la estructura de **decisión alternativa**
 - Usa las sentencia **if-else**
 - Condiciona la **ejecución de dos bloques** de sentencias a el cumplimiento o no cumplimiento de una **condición**
 - Si la condición se cumple (resulta **true**) se ejecuta el cuerpo de la sentencia **if**
 - Si la condición no se cumple (resulta **false**) se ejecuta el cuerpo de la sentencia **else**

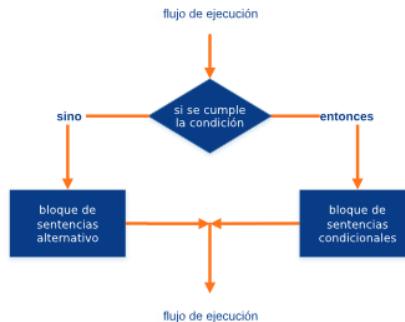
22



Decisión alternativa

- Sintaxis:

```
if <condición>:  
    <Bloque de sentencias condicionales>  
else:  
    <Bloque de sentencias alternativo>
```



23

Decisión múltiple

NEW



- Aprovechemos de ver otra estructura de decisión: la **decisión múltiple**

- Supongamos que necesitamos crear una función que calcule el impuesto a pagar por un ciudadano chileno dada su renta imponible anual de acuerdo a la siguiente tabla:

Banda	Desde (\$)	Hasta (\$)	Tasa impuesto	Descuento (\$)
0	0,00	6.513.372,00	0%	0,00
1	6.513.372,01	14.474.160,00	5%	325.668,60
2	14.474.160,01	24.123.600,00	10%	1.049.376,00
3	24.123.600,01	33.773.040,00	15%	2.255.556,60
4	33.773.040,01	Y MÁS	25%	5.632.860,60

24

Decisión múltiple

NEW



- Pensemos en la solución:
 - Existen **bandas** de renta con diferentes tasa de impuestos
 - Definiremos **constantes** para estos valores
 - La función necesita conocer la **renta imponible anual** del ciudadano (en pesos)
 - Un parámetro formal que llamaremos **renta**
 - La función ha de devolver el monto del **impuesto a pagar**
 - En pesos
 - De acuerdo a la banda
 - Cálculo es **renta * tasa de la banda – descuento de la banda**

25

Decisión múltiple

NEW



- El cálculo al interior de la función:
 - Suponemos constantes **LIMITE_BANDA_X** y **TASA_BANDA_X**
 - La banda cero (**renta < LIMITE_BANDA_1**) no paga impuestos y la función ha de devolver el valor cero
 - Cálculo:

```
Si renta >= LIMITE_BANDA_1 y renta < LIMITE_BANDA_2
    Devolver renta * TASA_BANDA_1 / 100.0
Sino, si renta >= LIMITE_BANDA_2 y renta < LIMITE_BANDA_3
    Devolver renta * TASA_BANDA_2 / 100.0
Sino
    Devolver renta * TASA_BANDA_3 / 100.0
```

26

Decisión múltiple

NEW



▪ El cálculo en más detalle:

- Suponemos constantes LIMITE_BANDA_X, TASA_BANDA_X y DESCUENTO_BANDA_X, con X = 1, 2, 3, 4
- La banda cero (`renta < LIMITE_BANDA_1`) no paga impuestos y la función ha de devolver el valor cero
- Cálculo usando decisiones alternativas:

```
Si renta >= LIMITE_BANDA_1 y renta < LIMITE_BANDA_2
    impuesto es renta * TASA_BANDA_1 / 100.0 - DESCUENTO_BANDA_1
Sino
    Si renta >= LIMITE_BANDA_2 y renta < LIMITE_BANDA_3
        impuesto es renta * TASA_BANDA_2 / 100.0 - DESCUENTO_BANDA_2
    Sino
        Si renta >= LIMITE_BANDA_3 y renta < LIMITE_BANDA_4
            impuesto es renta * TASA_BANDA_3 / 100.0 - DESCUENTO_BANDA_3
        Sino
            impuesto es renta * TASA_BANDA_4 / 100.0 - DESCUENTO_BANDA_4
```

Decisión múltiple

NEW



▪ Esta es una **decisión múltiple**

- Existen **varias condiciones**
- Pero **sólo una** se puede de cumplir simultáneamente
- Python ejecutará el bloque condicional de la **primera** condición que se cumple
- Si **ninguna condición** se cumple, Python ejecuta el **bloque alternativo (else)**
- Semánticamente, es sólo una forma abreviada de escribir **varias decisiones alternativas anidadas**

Decisión múltiple



NEW

- Sintaxis:

```
if <cond1>:  
    <Bloque de sentencias condicionadas a cond1>  
elif <cond2>:  
    <Bloque de sentencias condicionadas a cond2>  
elif <cond3>:  
    <Bloque de sentencias condicionadas a cond3>  
:  
elif <condn>:  
    <Bloque de sentencias condicionadas a condn>  
[else:  
    <Bloque de sentencias alternativo>]
```

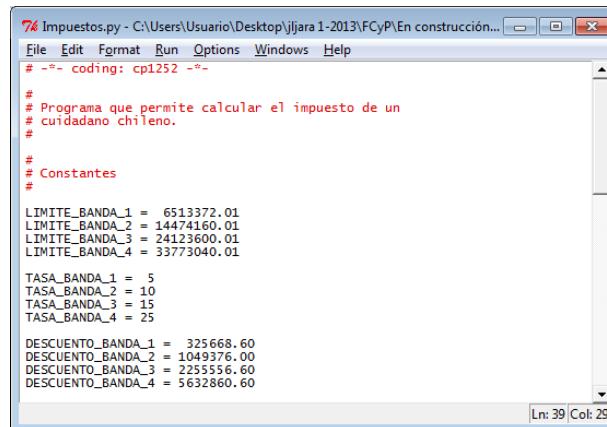
↑
opcional

29

Solución con decisión múltiple



- En la primera sección del programa definimos las constantes:



```
74 Impuestos.py - C:\Users\Usuario\Desktop\liljara 1-2013\FCyP\En construcción...  
File Edit Format Run Options Windows Help  
# -*- coding: cp1252 -*-  
  
#  
# Programa que permite calcular el impuesto de un  
# ciudadano chileno.  
#  
# Constantes  
#  
LIMITE_BANDA_1 = 6513372.01  
LIMITE_BANDA_2 = 14474160.01  
LIMITE_BANDA_3 = 24123600.01  
LIMITE_BANDA_4 = 33773040.01  
  
TASA_BANDA_1 = 5  
TASA_BANDA_2 = 10  
TASA_BANDA_3 = 15  
TASA_BANDA_4 = 25  
  
DESCUENTO_BANDA_1 = 325668.60  
DESCUENTO_BANDA_2 = 1049376.00  
DESCUENTO_BANDA_3 = 2255556.60  
DESCUENTO_BANDA_4 = 5632860.60
```

30



Solución con decisión múltiple

- La segunda sección del programa es la función que se nos pide:

```

# Impuestos.py - C:\Users\Usuario\Desktop\jijara 1-2013\FCyP\En construcción\I...
File Edit Format Run Options Windows Help
#
# Funciones
#
#
# Función que calcula el impuesto de un ciudadano chileno.
# Entrada: renta imponible anual en $
# Salida: impuesto a pagar en $ (valor entero)
#
def calculaImpuesto(renta):
    if renta < LIMITE_BANDA_1:
        return 0
    if renta >= LIMITE_BANDA_1 and renta < LIMITE_BANDA_2:
        impuesto = round(renta * TASA_BANDA_1 / 100.0 - DESCUENTO_BANDA_1)
    elif renta >= LIMITE_BANDA_2 and renta < LIMITE_BANDA_3:
        impuesto = round(renta * TASA_BANDA_2 / 100.0 - DESCUENTO_BANDA_2)
    elif renta >= LIMITE_BANDA_3 and renta < LIMITE_BANDA_4:
        impuesto = round(renta * TASA_BANDA_3 / 100.0 - DESCUENTO_BANDA_3)
    else:
        impuesto = round(renta * TASA_BANDA_4 / 100.0 - DESCUENTO_BANDA_4)
    return int(impuesto)

```

Ln: 39 Col: 29

31

Solución con decisión múltiple



- Función que usamos en el bloque principal del programa:

```

# Impuestos.py - C:\Users\Usuario\Desktop\jijara 1-2013\FCyP\En construcción\I...
File Edit Format Run Options Windows Help
#
# Bloque principal
#
#
# Entrada de datos
rentaImpAnual = input("Ingrese su renta imponible anual ($ chilenos): ")
# Procesamiento
impuestoAPagar = calculaImpuesto(rentaImpAnual)
# Salida de datos
print "Ud. debe pagar", impuestoAPagar, "pesos en impuestos"

```

Ln: 66 Col: 47

32



¿CONSULTAS?

33