


UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
Fundamentos de Computación y Programación (10110-1)



CLASE N°11

ITERACIÓN Y RECURSIÓN RELOADED

EXPRESIONES



- Hemos conocido **diversas herramientas** para poder resolver problemas programando:
 - Operaciones **aritméticas**
 - +, -, *, /, %, **
 - Con tres **tipos de números**
 - int: 2, -230, 45, -565343, etc
 - long: 2L, -2L, -205L, 1024L, 2147483648L
 - float (64 bits): 2.1, -2.56, -3.234e-2

2

EXPRESIONES



- También:
 - Operaciones **lógicas**
 - Comparaciones: ==, >, >=, <, <=, !=
 - Operadores lógicos: not, and, or
 - Dos posibles **valores booleanos**
 - True, False
 - **Expresiones booleanas** permiten **condicionar** la ejecución de secciones de código

3

TOMANDO DECISIONES



- Decisión simple

```
if <condición>:
    <Bloque de sentencias condicionales>
```
- Decisión alternativa

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

4

TOMANDO DECISIONES



- Decisión múltiple:

```
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

5


MEMORIA Y FUNCIONES



- Podemos **recordar valores** asignándolos a **variables** y **constantes**
- Tenemos disponibles un conjunto de **funciones nativas** para ser usadas
- Podemos **extender** este conjunto de funciones
 - **Importando** funciones desde módulos
 - **Creando** nuevas funciones

6

FUNCIONES



- Primero debemos **definir** la función


```
def entregaMayor(x, y):  
    if x < y :  
        return y  
    else :  
        return x
```

Diagram illustrating the structure of a function definition:

- Encabezado** (Header): The line starting with `def`.
- Nombre** (Name): The function name, `entregaMayor`.
- Parámetros formales** (Formal parameters): The parameters in parentheses, `(x, y)`.
- Cuerpo** (Body): The indented lines of code following the header.
- Líneas indentadas** (Indented lines): The lines of code within the function body.
- Línea en blanco sin indentar** (Unindented blank line): The blank line at the end of the function definition.
- Sentencias de retorno** (Return statements): The `return` statements within the body.

7

FUNCIONES




- Para **ejecutar** la función, debemos **invocarla** en una expresión
 - Con su **nombre**
 - Entregando un **parámetro actual** para cada parámetro formal
 - Ejemplo:

```
mayor = entregaMayor(5, valor)
```

8


BUENAS PRÁCTICAS



- A medida que los programas se alargan, se hacen **más difíciles de entender** por seres humanos y debemos preocuparnos de su **legibilidad**
 - Debemos seguir **buenas prácticas** de programación
 - Como mantener **secciones de código** separadas
 - Encabezado, importación y definición de constantes, importación y definición de funciones, bloque principal del programa
 - Usando **comentarios** para marcar cada sección
 - Usando **comentarios** para explicar qué hace cada función definida

9

BUENAS PRÁCTICAS



- 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

10

ESTRUCTURA DE PROGRAMAS



```
# ENCABEZADO
    # AUTOR, FECHA, OBJETIVO DEL PROGRAMA

# CONSTANTES
    # IMPORTACIÓN DE CONSTANTES
    # DEFINICIÓN DE CONSTANTES

# FUNCIONES
    # IMPORTACIÓN DE FUNCIONES
    # DEFINICIÓN DE FUNCIONES
```

11

ESTRUCTURA DE PROGRAMAS



```
# BLOQUE PRINCIPAL
    # ENTRADA DE DATOS
        # AQUÍ USE variable = INPUT()

    # PROCESAMIENTO
        # AQUÍ LLAME A LAS FUNCIONES

    # SALIDA
        # AQUÍ PONGA SUS SENTENCIAS PRINT
# FIN
```

12

BUENAS PRÁCTICAS



- Los **buenos identificadores** ahorran 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()`

13

REPETICIONES



- Podemos hacer que un conjunto de sentencias se **repita**
- Existen **dos métodos** para conseguir repeticiones:
 - **Iteración**
 - **Recursión**

14

ITERACIÓN



- Iteraciones de sentencias se consiguen mediante el **ciclo while**
- Debemos indicar **explícitamente** las sentencias a repetir dentro del **cuerpo** del **while**
- Estas sentencias se repiten **mientras** la expresión booleana que condiciona el **while** sea **verdadera**
- Se debe tener especial cuidado de no poner condiciones **siempre verdaderas (tautologías)** en un ciclo **while**

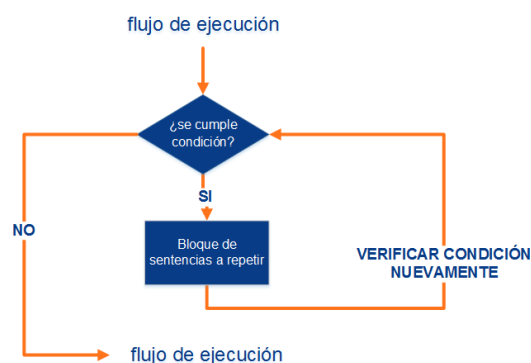
15

ITERACIÓN



- Sintaxis:

```
while <condición>:  
    <Bloque de sentencias a repetir>
```



16

RECURSIÓN



- El **cuerpo** de una función se puede **repetir** mediante el uso de **llamadas recursivas**
 - Una llamada recursiva es cuando una función se **llama a sí misma**, cambiando los parámetros actuales
- La repetición se realiza **implícitamente**, sin declarar exactamente qué queremos repetir
- Una llamada recursiva crea una **nueva instancia de ejecución** de la función, independiente de las otras

17

RECURSIÓN



- La repetición se detiene cuando se alcanza un **caso base**, es decir, cuando se alcanza **una instancia conocida** del problema
- Es importante asegurar que la función alcance el caso base **siempre**, o se ejecutará hasta consumir toda la memoria disponible
- Es una solución más elegante que el `while`, pero en general **consume más recursos** y es **menos eficiente**

18

RECURSIÓN



- Para **modelar** una función recursiva es mejor pensarla en términos de **dos partes**:
 - **Regla(s) de recursión**: Llamado(s) de la función a sí misma
 - **Condición de borde**: Detiene la recursión usando **el caso base**

```
n * factorial(n - 1)
```

```
if n == 0:  
    return 1
```

19

RECURSIÓN



- Con esto podemos construir la función recursiva:

```
def factorial(n)  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```



- Veamos un detalle:

NEW

- Esta función **falla** (recursión infinita) si se invoca con un parámetro formal que es **negativo** o **un flotante**

20

VALIDANDO PARÁMETROS





- Nos podemos “**asegurar**” que un parámetro actual es válido por medio del comando **assert**
- Sintaxis:

```
assert <expresión>, <mensaje al usuario>
```
- Semántica:
 - Si <expresión> resulta **falsa**, entonces se **detiene la ejecución** del programa por un **error de verificación** (AssertionError) con el **mensaje** indicado

21

VALIDANDO PARÁMETROS



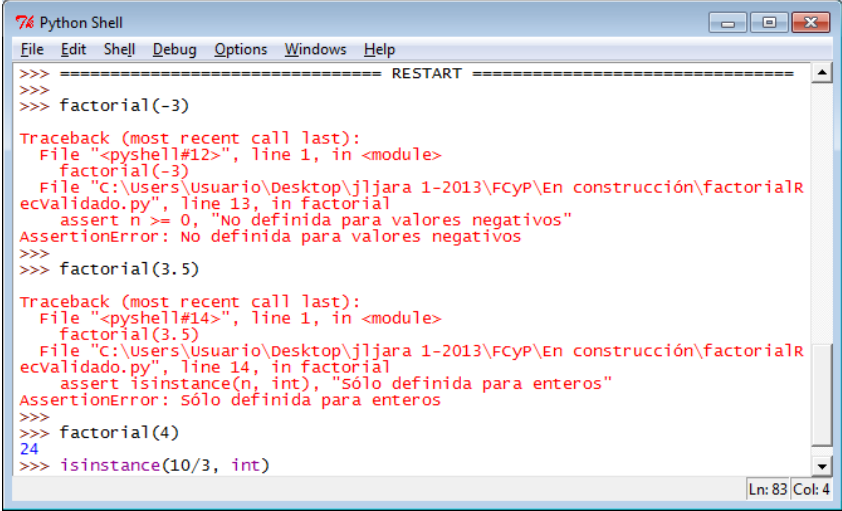
- Ahora podemos **mejorar** nuestra función **factorial()** recursiva:

```
def factorial(n)
    assert n >= 0, "No definida para valores negativos"
    assert isinstance(n, int), "Sólo definida para enteros"
    if n == 0:
        return 1
    return n * factorial(n - 1)
```
- Veamos su funcionamiento

22

VALIDANDO PARÁMETROS

NEW



```
>>> ===== RESTART =====
>>>
>>> factorial(-3)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    factorial(-3)
  File "C:\Users\Usuario\Desktop\jlljara 1-2013\FCyP\En construcción\factorialR
ecvalidado.py", line 13, in factorial
    assert n >= 0, "No definida para valores negativos"
AssertionError: No definida para valores negativos
>>> factorial(3.5)
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    factorial(3.5)
  File "C:\Users\Usuario\Desktop\jlljara 1-2013\FCyP\En construcción\factorialR
ecvalidado.py", line 14, in factorial
    assert isinstance(n, int), "Sólo definida para enteros"
AssertionError: Sólo definida para enteros
>>> factorial(4)
24
>>> isinstance(10/3, int)
```

Ln: 83 Col: 4

23

VALIDANDO PARÁMETROS

NEW

- Hacemos uso de la **función nativa isinstance()**
- Sintaxis:

`isinstance(<expresión>, <tipo>)`
- Semántica:
 - Retorna **True** si la expresión resulta del tipo indicado;
 - En caso contrario retorna **False**

24

