Ramificación e Iteración.

MIT 6.0001. Introduction to Computer Science and Programming in Python.

Resumen

Al programar, comúnmente estableceremos pruebas y repetiremos una o varias instrucciones. Estas tareas se agrupan en las **Estructuras de Control** y es en lo que nos concentraremos la mayor parte de esta clase. Al inicio revisaremos los datos de tipo "cadena" y las operaciones que podemos realizar con ellos. Luego, estudiaremos las declaraciones `if`, `else` y `elif`, así como los búcles (*loops*) `while` y `for`.

1. Datos de Tipo Cadena.

La clase anterior revisamos algunos objetos escalares, como los `int`, `floats`, `bool` y `NoneType`. Otro que es parte de ellos, es el de tipo **cadena** o `str`.

Los objetos de tipo `str` son una **secuencia de caracteres** las cuales pueden ser letras, caracteres especiales (como las tildes, comillas, etc) espacios y dígitos. En Python se los declara usando **comillas** (dobles o individuales) al inicio y al final de una cadena, como vemos en el siguiente ejemplo.

```
>>> saludo = "¡Hola a todos!"
>>> type(saludo)
<class 'str'>
```

Los valores de la variable `saludo` son `¡`, `H`, `o`, `l`, `a`, `espacio`, `a`, `espacio`, `t`, `o`, `d`, `o`, `s`, `!`.

1.1 Operaciones usando cadenas de caracteres.

Una de las operaciones más comunes con cadenas de caracteres, es la **concatenación**, la cual es la **unión de caracteres** mediante el operador `+`.

En el siguiente ejemplo definimos las variables `saludo` y `nombre` como dos cadenas de caracteres y las concatenamos para escribir un mensaje.

```
>>> saludo = "hola"
>>> nombre = "Ricardo"
>>> saludo + " " + nombre
'hola Ricardo'
```

Agregamos explícitamente un espacio mediante la expresión `" "` porque Python no lo añade por sí mismo, como lo vemos a continuación.

```
>>> saludo + nombre
'holaRicardo'
```

La única operación que podemos realizar **entre cadenas y números** en Python, es la **multiplicación** `*`. Básicamente, al agregar un producto al lado derecho de un caracter significa que lo **estamos repitiendo** por una cantidad de veces determinada por nosotros.

```
>>> saludo*3 + " " + nombre
'holaholahola Ricardo'
```

1.2 Concatenando adentro de la función `print`.

La clase anterior vimos que la función `print()` nos permite "imprimir" objetos en la terminal. En ella, también es posible realizar **concatenaciones**.

A continuación tenemos un programa que llamaremos `numero_favorito.py`, el cual imprime el número que más nos gusta. Para este ejemplo, digamos que dicho valor es `1`, el cual lo almacenaremos en una variable llamada `x`.

```
#!/usr/bin/env python3
# Concatenando usando la función print.
# Ejemplo 1.
x = 1
print("mi número favorito es", x, ".", "Gracias por leer (:")
```

Al ejecutar el script obtenemos lo siguiente[1]:

```
$ python numero_favorito.py
Mi número favorito es 1 . Gracias por leer (:
```

[1]Cuando en un bloque de código uso el símbolo `$` quiere decir que estoy ejecutando desde una shell (usando o no una terminal).

Como se puede apreciar, usamos los símbolos `,` (comas) para concatenar, los cuales añaden automáticamente los espacios.

No obstante, quizá hay algunos caracteres en los cuales no queremos algunos espacios (e.g, en el punto seguido de la oración vista arriba). En ese caso podemos usar los operadores `+` para concatenar adentro de la función `print()`. Algo a tener en consideración es que si incluimos valores numéricos en la cadena, antes debemos transformarlos a `str` usando la función `str()`.

```
#!/usr/bin/env python3
# Concatenando usando la función print.
# Ejemplo 2.
x = 1
print("Mi número favorito es " + str(x) + ". " + "Gracias por leer (:")
```

Ejecutemos el script.

```
$ python numero_favorito.py
Mi número favorito es 1. Gracias por leer (:
```

En definitiva, si en la función `print()` usamos comas para concatenar, no es necesario que todos los objetos sean cadenas, pero si usamos el operador `+` entonces todos deben serlo.

1.3 Interactuando con la consola usando la función `input`.

Para crear un programa en el cual sea posible interactuar con la consola, usamos la función `input()`. Cuando ingresamos un valor en su argumento, al ejecutarla espera en la consola que agreguemos otro. Al hacerlo, este último se convierte en una cadena de caracteres y termina.

Como el valor de salida de la función `input()` es una cadena, si queremos usarlo para, digamos, una operación matemática, debemos transformarlo a un valor numérico (i.e, `int` o `float`). Por ejemplo, a continuación tenemos un programa llamado `tabla_del_8.py` que calculará el producto entre el número `8` y otro que ingresamos de forma interactiva.

```
#!/usr/bin/env python3
# El caracter `\n` agrega una nueva línea.
numero = input("Ingresa el número que multiplicará a 8.\nNúmero: ")
producto = 8*int(numero)
print("8 x", numero, "=", producto)
```

Ejecutémoslo para calcular el producto entre 8 y 13.

```
$ python tabla_del_8.py
```

```
Ingresa el número que multiplicará a 8.
Número: 13
8 x 13 = 104
```

2. Estructuras de Control.

Hasta ahora hemos visto programas que se ejecutan linealmente, pero como vimos la clase anterior es posible controlar en qué ocasiones se ejecutará una o más instrucciones, o la cantidad de veces en que se realizará una tarea. El conjunto de declaraciones que nos permiten hacer aquello reciben el nombre de **estructuras de control** (*control flow*).

2.1 Operadores de Comparación.

En programación, el control de flujo se realiza principalmente usando **operadores de comparación**. En esta clase nos centraremos en aplicarlos a datos de tipo `int`, `float` y `str`.

A continuación tenemos una tabla donde aparecen los operadores de comparación y la descripción de cada uno, donde `i` y `j` son nombres de variables.

Operador	Descripción
`i > j`	`i` mayor a `j`
`i >= j`	`i` mayor o igual a `j`
`i < j`	`i` menor a `j`
`i <= j`	`i` menor o igual a `j`
`i == j`	`i` es igual a `j` (prueba de igualdad)
`i != j`	`i` no es igual a `j` (prueba de desigualdad)

Cuando evaluamos dos o más objetos con operadores de comparación, obtenemos un **valor de tipo booleano** que en Python se denota como `bool` y corresponden a `True` y `False`.

```
>>> 2 < 5
True
>>> type(2 < 5)
<class 'bool'>
```

Algunos tipos de datos pueden compararse con si mismos y con otros, pero no todos tienen esa característica. Para los casos de `int`, `float` y `str` los tenemos en la siguiente tabla.

Tipo de dato	Comparaciones posibles
`int`	`int`, `float`
`float`	`float`, `int`
`str`	`str`

Como vemos, un valor de tipo `str` solo puede compararse con otro de su mismo tipo, en caso contrario en Python se genera un error de semántica estática.

```
>>> "b" > 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

Otra característica de los datos de tipo `int`, es que la comparación entre caracteres usando los operadores `>` o `<` está determinado según su **orden alfabético**, donde el menor de todos es la letra `a` y el mayor es `z`.

```
>>> "a" < "z"
True
>>> "f" > "k"
False
```

2.2 Operadores Lógicos.

También es posible comparar dos o más valores de tipo `bool` por medio de **operadores lógicos**, los cuales son: `and`, `or` y `not`. A continuación tenemos dos ejemplos:

```
>>> True and False
False
>>> not (2 > 5)
True
```

Como vemos, con los operadores lógicos comparamos dos o más valores booleanos y obtenemos uno nuevo del mismo tipo.

En la siguiente **tabla de verdad** vemos las combinaciones posibles con operadores lógicos.

Sean `a` y `b` dos valores booleanos:

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

En cuanto al operador `not` tenemos la siguiente tabla de verdad:

a	not a
True	False
False	True

2.3 Ramificando un Programa (*branching*).

Con la ayuda de los operadores de comparación y lógicos podemos construir pruebas para determinar si el programa ejecuta o no un conjunto de instrucciones, **ramificándolo**. Para ello, usamos **declaraciones condicionales** que en Python corresponden a los comandos `if`, `else` y `elif`.

En general, una declaración condicional comienza con el comando `if` más la prueba dada como una expresión booleana y un **bloque de código** que son las instrucciones que queremos que se ejecutarán si se cumple el test (i.e, si el valor booleano resultante es `True`). La estructura en Python es como la que vemos a continuación:

```
if <prueba>:
  <expresión 1>
  <expresión 2>
  ...
```

Si la prueba no se cumple (i.e, si su valor booleano es `False`), el bloque de código adentro de la declaración `if` no se ejecutará y el programa continuará con las instrucciones que están posterior a ésta o se terminará.

Ahora, también puede ser nuestra intención que el programa ejecute otro conjunto de instrucciones en caso de que no se cumpla la prueba de la declaración `if`. Para aquello, añadimos el comando `else` más otro bloque de código.

```
if <prueba>:
  <expresión 1>
  <expresión 2>
  ...
else:
  <expresión 1>
  <expresión 2>
  ...
```

Por otra parte, también es posible hacer que el programa ejecute otro bloque de código si no se cumplen las de `if`, pero estableciendo otras condiciones a la vez. En dicho caso, usamos la declaración `elif`, que es una forma acotada de escribir "else if".

```
if <prueba>:
  <expresión 1>
  <expresión 2>
  ...
elif <prueba>:
  <expresión 1>
  <expresión 2>
  ...
else:
  <expresión 1>
  <expresión 2>
  ...
```

Veamos que la declaración `else` siempre se mantiene al final, porque es la que se ejecuta cuando ninguna prueba se cumple. Es decir, en el caso donde la de `if` como la de `elif` son `False`.

Entonces, cuando queremos establecer declaraciones condicionales en nuestro código, el comando `if` siempre debe estar, mientras que `elif` y `else` son **opcionales**. En otras palabras, si estas dos últimas no están, el programa se ejecutará igual y sin errores.

A continuación tenemos programa que llamamos `branching_example.py` que evalúa si el número que ingresamos es positivo, negativo o cero, así como si es par o impar. Como podemos

observar, es posible agregar o **anidar** (*nest*) declaraciones condicionales adentro de otras.

```
#!/usr/bin/env python3
numero = int(input("Ingresa un número: "))

if numero > 0:
  if (numero % 2) == 0:
    print("Tu número es positivo y par")
  else:
    print("Tu número es positivo e impar")
elif numero < 0:
  if (numero % 2) == 0:
    print("Tu número es negativo y par")
  else:
    print("Tu número es negativo e impar")
else:
  print("Tu número es cero")

print("Gracias (:")
```

Ejecutemos este script para el número `-3`.

```
$ python branching_example.py
Ingresa un número: -3
Tu número es negativo e impar
Gracias (:
```

Algo a tener en cuenta al programar en Python, es que cuando escribimos un bloque de código, por ejemplo, adentro de una declaración condicional, debemos **indentar** o agregar **al menos un espacio** antes de cada expresión. En caso contrario, se genera un error de sintáxis.

Por otra parte, no debemos olvidar los `:` al final de cada `if`, `else` o `elif`. También es parte de la sintáxis de Python.

2.4 Iteraciones.

En ocasiones, sea escribiendo un programa de forma lineal o ramificándolo, vamos a necesitar repetir una tarea más de una vez para un mismo valor o para otros.

Por ejemplo, digamos que queremos desarrollar un script que nos entregue la tabla de multiplicación que queramos conocer, desde 1 hasta 12. De forma lineal podríamos escribirlo de la siguiente manera:

```
#!/usr/bin/env python3
tabla = int(input("¿Qué tabla buscas calcular?\nIngresa el número: "))

print("\nTabla del", tabla, "\n------------")

print(tabla, "x", 1, "=", tabla*1)
print(tabla, "x", 2, "=", tabla*2)
print(tabla, "x", 3, "=", tabla*3)
print(tabla, "x", 4, "=", tabla*4)
print(tabla, "x", 5, "=", tabla*5)
print(tabla, "x", 6, "=", tabla*6)
print(tabla, "x", 7, "=", tabla*7)
print(tabla, "x", 8, "=", tabla*8)
print(tabla, "x", 9, "=", tabla*9)
print(tabla, "x", 10, "=", tabla*10)
print(tabla, "x", 11, "=", tabla*11)
print(tabla, "x", 12, "=", tabla*12)
```

Nombremos al script como `iteration_1.py`. Ejecutémoslo para conocer la tabla de multiplicación del 3.

```
$ python iteration_1.py
¿Qué tabla buscas calcular?
Ingresa el número: 3

Tabla del 3
------------
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
```

```
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
3 x 11 = 33
3 x 12 = 36
```

Otra manera de escribir el código en `iteration_1.py`, es ramificándolo usando declaraciones condicionales. En el script `iteration_2.py` añado un contador el cual uso como el factor que multiplica al número de la tabla. A medida que se cumple cada prueba, éste aumenta en una unidad. Por temas de espacio, solo pondré una parte del código.

```
#!/usr/bin/env python3
tabla = int(input("¿Qué tabla buscas calcular?\nIngresa el número: "))
contador = 1

print("\nTabla del", tabla, "\n------------")

if contador == 1:
  print(tabla, "x", contador, "=", tabla*contador)
  contador = contador + 1

if contador == 2:
  print(tabla, "x", contador, "=", tabla*contador)
  contador = contador + 1
...
```

Al ejecutar `iteration_2.py` para el número 3 obtenemos la misma tabla de `iteration_1.py`.

2.4.1 Búcle `while`.

El problema en los scripts de `iteration_1.py` y `iteration_2.py` es la cantidad de líneas de código escritas de una misma tarea, pero para un valor distinto. En general, nuestro ideal siempre será minimizar lo más posible este número, porque permite que un programa se ejecute más rápido[2]. En este caso, podemos reducir aquello usando **Búcles** (*Loops*).

Los **búcles** son **mecanismos de iteración** (o repetición) de un conjunto de instrucciones las cuales **se aplican hasta que se cumple una condición**. En Python, uno de ellos se llama `while` y su estructura es la siguiente:

[2]El tiempo de ejecución de un programa está dado por la cantidad de pasos que debe seguir.

```
while <condición>:
  <expresión 1>
  <expresión 2>
  ...
```

Como vemos, es similar a la de las declaraciones condicionales, salvo por el hecho de que el bloque de código adentro de `while` se ejecuta sí o sí hasta que se cumple la `<condición>`.

En el siguiente script llamado `iteration_3.py` reescribimos el programa inicial de `iteration_1.py` usando un búcle `while`.

```
#!/usr/bin/env python3
tabla = int(input("¿Qué tabla buscas calcular?\nIngresa el número: "))
contador = 1

print("\nTabla del", tabla, "\n------------")

while contador <= 12:
  producto = tabla*contador
  print(tabla, "x", contador, "=", producto)
  contador += 1 # Es lo mismo a `contador = contador + 1`

print("\n¡Gracias!")
```

Al igual que en `iteration_2.py`, en `iteration_3.py` establecemos un contador que incializamos al principio, pero éste va aumentando adentro del búcle `while` y las 2 primeras líneas al interior de éste se ejecutan mientras `contador <= 12`. Cuando sea mayor `12`, se detiene.

En un búcle `while` como el que escribimos en `iteration_3.py` tenemos que añadir un contador en su condición y que vaya aumentando o disminuyendo medida que se ejecuta hasta que se cumpla, porque podríamos entrar a un **búcle infinito** (*infinite loop*). Esto puede ocurrir si borramos el comando `contador += 1`, ya que `contador` siempre será menor a `12`. Si ocurre, para detenerlo simplemente presionamos las teclas `CTRL + c`.

2.4.2 Búcle `for`.

El otro búcle que existe en Python es `for`, el cual podemos entenderlo como una forma resumida de `while`. Su estructura es como la que sigue:

```
for <variable> in <longitud_de_secuencia>:
  <expresión 1>
  <expresión 2>
  ...
```

El bucle `for` es similar a `while` en su propósito de repetir una tarea, pero lo hace considerando el **largo de una secuencia** y no una condición de término. En ese sentido, el **primero** es recomendado cuando **conocemos la cantidad de iteraciones que queremos realizar** y el **segundo** en el caso en que ésta es **ilimitada**.

Usemos el búcle `for` en un nuevo script llamado `iteration_4.py` que también crea una tabla de multiplicación. Para la longitud de la secuencia utilizaremos la función `range()` la cual generará números en el intervalo $[1,\ 13)$ y avanzan en una unidad.

```
#!/usr/bin/env python3
tabla = int(input("¿Qué tabla buscas calcular?\nIngresa el número: "))

print("\nTabla del", tabla, "\n------------")

for contador in range(1, 13):
  producto = tabla*contador
  print(tabla, "x", contador, "=", producto)
```

Como vemos, la variable `contador` va tomando cada valor creado por `range()` y va avanzando de forma secuencial según la cantidad de elementos que genera aquella función, que en este caso son 12.

La función `range()` suele ser usada para generar una secuencia en un búcle `for`. Requiere que ingresemos como argumento al menos un objeto `int`, el cual indica el valor de término que, en nuestro ejemplo, fue `13`. También es opcional agregar al principio un valor de inicio (por defecto es `0`) y al final en cuántas cantidades irá avanzando (por defecto es `1`). En nuestro caso mantuvimos este último sin cambios y el primero que comenzara en `1`.

2.4.3 Declaración `break` adentro de un búcle.

En algunos lenguajes de programación, como en Python, tenemos la opción de **detener un búcle antes de que termine si se cumple una o varias condiciones**. En este último podemos lograr aquello usando la **declaración** `break`.

Por ejemplo, en el siguiente script llamado `iteration_break.py` a partir de un búcle `while`

vamos a imprimir 101 números, desde 100 hasta 0, pero se detendrá en el primer múltiplo de 6 que encuentre.

```
#!/usr/bin/env python3
contador = 100

while contador >= 0:
  print(contador)
  contador -= 1
  if contador % 6 == 0:
    print("¡Detente! --> " + str(contador) + " es múltiplo de 6.")
    break
```

Al ejecutar `iteration_break.py` obtenemos lo siguiente:

```
$ python iteration_break.py
100
99
98
97
¡Detente! --> 96 es múltiplo de 6.
```