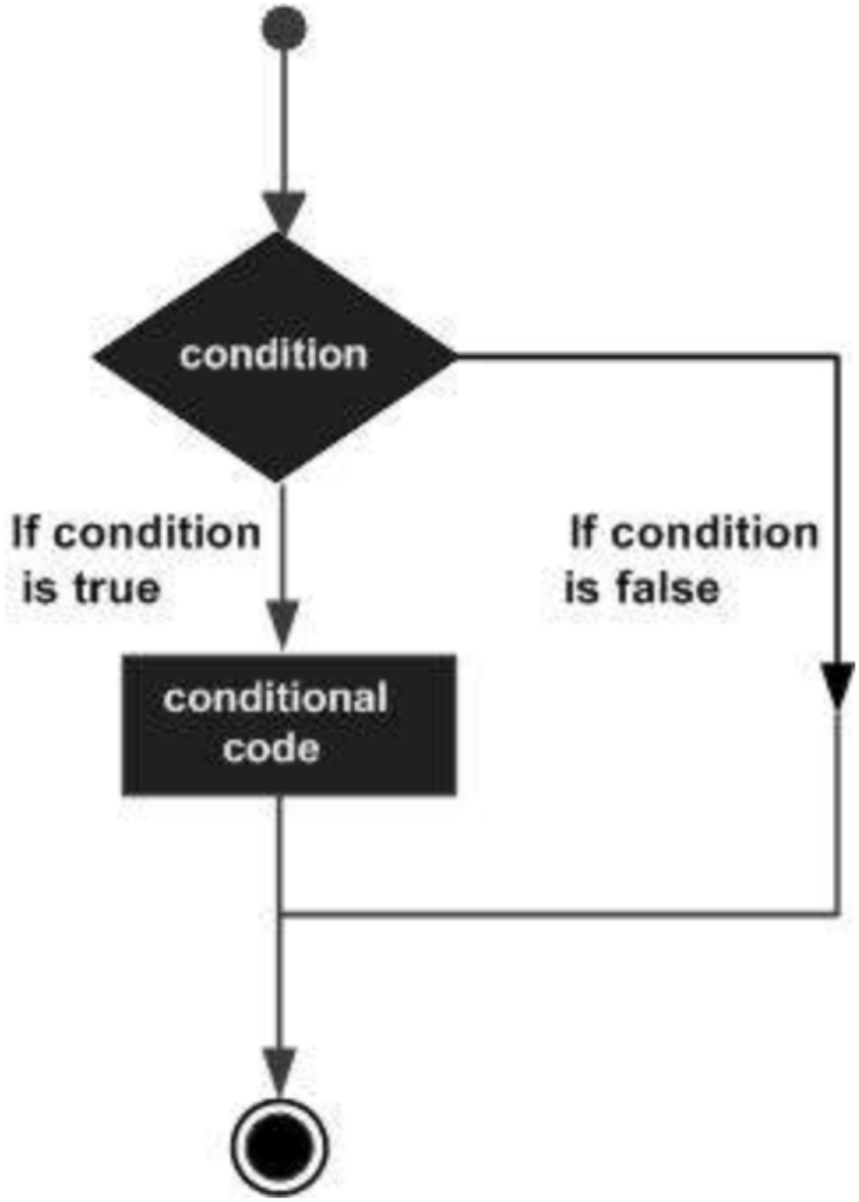
[Python] Cheat Sheet 2

Este documento contiene información que orienta al lector para su entendimiento de las estructuras de flujo que se utilizan en el lenguaje Python. tenga en cuenta que no es una lista completa ni atenta serlo.

Condicional:

La estructura de control de flujo por default en el lenguaje es el condicional. Dicha estructura viene a representar el siguiente diagrama de flujo:



Source: https://www.tutorialspoint.com/python/python_decision_making.htm

IF:

El condicional más simple es el simple `if` el cual plantea que un bloque de código se ejecuta solo si se cumple una condición.

El siguiente fragmento muestra el uso básico de dicha estructura:

```
# Complete If statemente

if condition:
  '''
  Do stuff here ...
  '''
```

En este ejemplo vemos el uso de un simple bloque `if` , dicha estructura comienza por el uso de la palabra reservada `if` seguida de una condición* que arroje algún valor de verdad (de tipo `bool`) y finalmente el uso de `:` para simboliza que termina la definición de la condición a utilizar. Luego, en la linea siguiente (puede dejarse la cantidad de lineas en blanco que se deseen) debe venir un `bloque` ** de código indentado con 4 espacios desde el nivel de indentación que tenga el respectivo `if` .

* La condición es cualquier fragmento de código Python válido que al computarlo termine dando un valor bool. Esto incluye pero no se limita a :
Ej:
→ valores de tipo Bool
→ Operaciones lógicas (comparación, buscar por el igual, etc)
→ retornos de función

** un `bloque` de código se considera a cualquier secuencia de código válida.
Ej:
`var = 'text'`
`print("hola")`
`break`
Son secuencias válidas, como todas las ya vistas en clase.

El valor `None` se evalúa como un False dentro de Python. Mientras que la presencia de cualquier elemento distinto de `None` (`int` , `float` , colleciones, etc) es considerada como `True` . Esto es por el modo de implementación que tiene el lenguaje y no se recomienda su uso.

IF - ELSE :

Vimos el uso de un simple condicional, donde si la condición se cumple el bloque de código se ejecuta pero en caso contrario el programa sigue de largo. Hay casos donde depende del valor de una condición debemos tomar una decisión que si o si modifica el flujo del programa. Estos casos hace uso del `if-else` statement.

El siguiente fragmento muestra su uso:

```
# If else
if condition:
  '''
  Do stuff here if condition is True
  '''
else:
  '''
  Do stuff here if condition is False
  '''
```

```
'''
The code continues here ...
'''
```

En este fragmento vemos el uso de la palabra reservada `if` seguida del predicado `condition` * y luego el uso de los `:` para simbolizar que la condición de decisión finalizó. El bloque que corresponde al código que debe ejecutarse en caso de que dicha condición se cumpla será el primer bloque **indentado** que el sistema encuentre. Luego debe utilizarse la palabra reservada `else` seguida de los `:` para indicar que a continuación se encuentra el bloque de código que corresponde al caso que la condición se falsa. Este segundo bloque debe tener el mismo nivel de indentación que el que se utilizó para especificar el bloque de la condición positiva.

ELIF:

cuando queremos evaluar multiples condiciones las cuales son mutuamente excluyentes, se recomienda el uso del `elif` . En este caso veamos un ejemplo a modo de clarificar la idea:

Si tenemos que imprimir la relación de un número con respecto a un intervalo dado que:

→ Si n es menor que el limite inferior, imprimir "Es inferior"

→ Si n está dentro de los límites, imprimir "Incluido"

→ Si n es superior al límite superior, imprimir "Es superior"

```
# elif example:
n = int(input("ingrese un numero"))
min_interval = 18
max_interval = 100

if n < min_interval:
  print("Es inferior")
elif n > max_interval:
  print("Es superior")
else:
  print("Incluido")
```

⚠️ Notar que en el ejemplo, el uso de `elif` nos obliga a prestar especial atención en el orden de las condiciones. Dado que el mismo puede alterar el resultado final de la ejecución.

Iteración:

La segunda estructura muy utilizada del lenguaje es la iteración sobre algún tipo de colección o condición. En este caso veremos que Python nos permite el uso de 2 estructuras para este objetivo.

FOR:

El primer elemento que veremos es el `for` dicha estructura tiene como objetivo iterar sobre una colección limitada de elementos sobre los que se quiere aplicar alguna transformación o se los quiere utilizar para operar.

La estructura es la siguiente:

```
# For
for loop_item in collection:
  '''
  Code which executes one for each element in the collection.
  take into account that the value of loop_item will change in each iteration to a different element.
  '''
```

En este caso vemos que la estructura comienza con el uso de la palabra reservada `for` seguido de la definición de una **variable de iteración***, luego se indica la palabra reservada `in` para indicar el conjunto de iteración (collection) y finalmente el uso de `:` para indicar que terminó la definición del conjunto.

El bloque de código a ejecutar por cada iteración debe estar indentado 4 espacio desde el nivel de indentación de la palabra `for` . Es importante remarcar que dicho código se va a ejecutar una vez por cada elemento del conjunto collection, en cada ciclo la variable loop_item tomará un valor nuevo del conjunto.

⚠️ * la variable de iteración toma la asignación por el sistema al comienzo del bloque. Es decir que en cada **loop** ya tiene un valor asignado de la colección. Esta variable es de scope local al `for` y al finalizar cada loop la misma es reasignada por el propio sistema con lo cual no debemos hacernos cargo.

🗣️ Debemos entender que el bloque de iteración se ejecuta una vez por cada elemento del conjunto de iteración. El valor de loop_item y como el mismo se modificará depende el tipo de conjunto, en caso de listas se iteran secuencialmente de izquierda a derecha pero los sets no tienen orden estipulado.

💡 la palabra reservada `in` se utiliza para indicar que una variable o valor se encuentra dentro de un conjunto.
su uso en condiciones sería:
`if 1 in (1, 2, 3, ):`

While:

La otra estructura que Python nos deja disponibles para iterar es el `while` dicha estructura tiene uso cuando nos interesa ejecutar un ciclo según el valor de una condición que conocemos o para ciclos infinitos.

Su estructura es la siguiente:

```
# While
while condition:
  '''
  Do stuff here
  '''
```

La estructura comienza con la palabra reservada `while` seguida de un predicado que se evalúe a un valor bool y finalmente el uso de los `:` para indicar el final de la condición.

Es importante entender que el predicado se evalúa antes de cada iteración. Es decir que solo se efectúa el ciclo si la condición es verdadera, en caso contrario omite todo el bloque de código indentado 4 espacios desde el nivel de indentación que tenga la palabra `while` .

💡 Cuando hablamos de predicado es el mismo concepto que la condición de un `if` .

⚠️ La estructura de `while` no nos asegura que termine de ejecutar en algún momento, ya que si la condición se mantiene verdadera el código seguirá ejecutando infinitas veces. Este problema ocurre frecuentemente cuando la ejecución del bloque de iteración no ejerce alguna modificación sobre aquello que se efectúa la condición.

Funciones [Parte I]:

En todo lenguaje de programación el concepto de funciones es primordial y permite acelerar y expandir el desarrollo de software a los niveles que están acostumbrados hoy en día.

su estructura es la siguiente:

```
# FUNCTIONS:

# example with arguemnts
def function_name(arg1, arg2 ...):
  '''
  Do stuff here
  '''
```

```
# example with return statement
def function_return(name):
  '''
  This function, expects name to be a string.
  It returns the value in upper case.
  '''
  return name.upper()


# example with no arguments
def greetings():
  print("hello World!")
```

En estos ejemplos vemos que la definición de una función comienza con el uso de la palabra reservada `def` seguido de un identificador válido que será la forma en que podemos llamar a la función, luego vienen los `(` y `)` entre los cuales pondremos los **argumentos** de la función y finalmente terminamos la estructura con el uso de `:` para indicar el comienzo del bloque de código que ejecuta al función.

Esencialmente las funciones son una forma de reutilizar un procedimiento multiples veces sin necesidad de reescribirlo. Esto está fuertemente ligado al concepto de **Modularización** que significa romper un problema complejo es pequeñas partes más simples que son fácilmente resolubles y así obtener una solución al problema principal.

Las funciones, como ya dijimos son una forma de condensar un bloque de código que queremos reutilizar en multiples ocaciones sin tener que reescribirlo. Las mismas tienen la particularidad de que a veces queremos o necesitamos que el procedimiento que vamos a aplicar sea dependiente de un dato que esta sujeto a cuando **aplicamos dicho procedimeinto.** Para expresa estas partes móviles del procedimiento que representa una función, utilizamos los **Argumentos**, que simplemente son los identificadores que mencionamos entre los `()` en al definición de la función. Estos identificadores son variables que se pueden utilizar dentro de la función tienen la particularidad de que van a tener el valor que nos indiquen cuando se **invoque** la función.

Invocación:

vimos que las funciones son una forma de empaquetar un código que utilizaremos muchas veces. Pero como todo paquete, debe tener una forma de recibir mensajes desde afuera y una forma de devolver los resultados que ejerce.

La primera parte, *recibir* se logra mediante el uso de los Argumentos. Si pensamos en las funciones definidas en el bloque anterior:

```
# invocation
# I want to print my name in uppercase
uppercase_name = function_return("emily")
print(uppercase_name)
# which after execution of the function will be
uppercase_name = "EMILIY"
```

```
print(uppercase_name)

# ====================================
# the above code could be translate into:
uppercase_name  = name.upper("emily")
print(uppercase_name)
```

En este ejemplo la estamos llamando a ejecutar el bloque de código definido bajo el identificador `function_return` , especificando que el valor de su argumento `name` en este caso será el string `"emily"` .

Ahora, dicha función ejecuta una operación (en este caso es pasar todas las letras a mayúscula) y nos interesa obtener ese resultado nuevamente. Para ello se hace uso de la palabra reservada `return` , esta indica que la función termina y que devuelve el valor indicado a continuación de `return` como resultado de su invocación.

El orden de los argumentos es importante a la hora de invocar una función!

Las funciones pueden tener multiples argumentos, los mismos en esos casos la invocación será de la siguiente forma:

```
def multiple_args(arg1, arg2, arg3):
  return None


# Multiple Args Inv

value_1 = 1
value_2 = 2
value_3 = 3

multiple_args(value_1, value_2, value_3)
```

En este caso, al invocar la función `multiple_args` tendrá definido `value_1` como valor del identificador `arg1` , `value_2` como valor del identificador `arg2` y `value_3` como valor del identificador `arg3`. Por esto es importante saber el orden de los parámetros de una función.

El nombre parámetro y argumento refieren a lo mismo, es el identificador que especificamos en la definición de una función.

En casos donde no sabemos el orden de los parámetros pero si el nombre de los mismos, podemos invocar la función forzando el orden de los parámetros:
Ej: `multiple_args(arg2=value_2, arg3=value_3, arg1=value_1)`
Este código ejecuta exactamente igual que el llamado del bloque anterior.

Argumentos:

Las funciones definidas por el usuario, en Python siempre toman el valor del argumento por **referencia** esto implica que el valor es modificado.

```
def fun(l):
  l.append(4)
  return l

my_list = [1,2, 3]
fun(my_list)
'''
After invocation
>>> my_list
>>> [1,,2, 3, 4]
'''
```

Los Argumentos pueden ser:

- Requeridos
- Default
- `kayword arguments`
- `variable-length arguments` ⚠️

Requeridos:

Son aquellos que en al definición de la función no tienen un valor por default y que deben ser especificados a la hora de invocar.

Ejemplo:

```
def my_fun(string):
  return None

my_fun()
'''
This code will produce an error because no value was assign to the paramenter 'string'
'''
```

Default:

En caso de que la definición de una función tenga valores por default para un argumento el mismo no es necesario especificar.

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print("Name: ", name)
   print("Age ", age)


# Now you can call printinfo function
printinfo( age=50, name="tony" )
printinfo( name="rose" )
```

Keyword Arguments:

Nos permite forzar a la asignación de los parámetros de una función sin importar del orden en que fueron escritos en su definición.

Ejemplo:

```
def my_fun(name, age):
  print(f"Hello {name} you are {} years old")

my_fun(age=50, name='santi')
```

⚠️ Una vez que se especificó un argumento del tipo `keyword` todos los demás argumentos deben ser especificados de la misma forma.

Scope:

Las variables pueden tener scope `local` o `global` y dependiendo de esto hay ocasiones donde no se puede acceder a ellas.

Ejemplo:

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

En el fragmento anterior la variable `total` es de scope `local` y no se puede acceder a la misma desde afuera de la función.