Programación para la Bioinformática

Módulo 2: Breve introducción a la programación en Python

Instrucciones de uso

A continuación se irá presentando la sintaxis básica del lenguaje de programación Python junto a ejemplos interactivos.

Variables y tipos de variables

Podemos entender una variable como un contenedor en el que podemos poner nuestros datos a fin de guardarlos y poderlos tratar más adelante. En Python, las variables no tienen tipo, es decir, no tenemos que indicar si la variable será numérica, un carácter, una cadena de caracteres o una lista, por ejemplo. Además, las variables pueden ser declaradas e inicializadas en cualquier momento, a diferencia de otros lenguajes de programación.

Para declarar una variable, utilizamos la expresión *nombre_de_variable = valor*. Se recomienda repasar la guía de estilo de Python, PEP-8 (https://www.python.org/dev/peps/pep-0008/), para indicar nombres de variables correctos, pero *grosso modo*:

- Evitaremos utilizar mayúscula en la letra inicial
- Separaremos las diferentes palabras con el carácter '_'
- No utilizaremos acentos ni caracteres específicos de nuestra condificación como el símbolo del '€' o la 'ñ', por ejemplo.
- Utilizaremos el carácter '#' para indicar que la línea se trata de un comentario.

Veamos unos cuantos ejemplos de declaraciones de variables y cómo usarlas:

In [4]:
```
# Esto es un comentario, por eso aparece en verde.

# Declaramos una variable de nombre 'variable_numerica'
# que contiene el valor entero 12
variable_numerica = 12

# Declaramos una variable de nombre monstruo que
# contiene el valor 'Godzilla'
monstruo = 'Godzilla'

# Declaramos una variable de nombre planeta que es una
# lista de cadenas de caracteres
planetas = ['Mercurio', 'Venus', 'Tierra', 'Marte']
```

Las variables nos permiten manipular los datos que contienen. El tipo de manipulación dependerá de los datos que contengan:

In [2]:
```
mi_edad = 25
mi_edad_en_5 = mi_edad + 5

# 'Imprimimos' el valor calculado que será, efectivamente, 30
print(mi_edad_en_5)
```

```
30
```

Los tipos nativos de datos que una variable de Python puede contener son: números enteros (int), números decimales (float), números complejos (complex), cadena de caracteres (string), listas (list), tuplas (tuple) y diccionarios (dict). Veamos uno por uno cada uno de estos tipos:

In [6]:
```
# Un número entero
int_var = 1
another_int_var = -5

# Podemos sumarlos, restarlos, multiplicarlos o dividirlos
print(int_var + another_int_var)
print(int_var - another_int_var)
print(int_var * another_int_var)

# Fijaos en esta última operación: se trata de una división entera.
# Como solo tratamos con números enteros, no debería tener parte decimal,
# pero las divisiones en Python 3 son por defecto en el espacio de los
# números decimales.
print(int_var / another_int_var)

# Si queremos dividir sin parte decimal, podemos utilizar esta versión:
print(int_var // another_int_var)
```

```
-4
6
-5
-0.2
-1
```

In [7]:
```
# Un número decimal o 'float'
float_var = 2.5
another_float_var = .7

# Convertimos un número entero en uno decimal
# mediante la función 'float()'
encore_float = float(7)
print(encore_float)

# Podemos hacer lo mismo en sentido contrario
# con la función 'int()'
new_int = int(encore_float)
print(new_int)
```

```
7.0
7
```

In [9]:
```
# Un número complejo
complex_var = 2+3j

# Podemos acceder a la parte imaginaria o a la parte real:
print(complex_var.imag)
print(complex_var.real)
print(complex_var)
```

```
3.0
2.0
(2+3j)
```

In [11]:
```
# Cadena de caracteres
my_string = 'Hello, Bio!'
print(my_string)

# Podemos escribir caracteres según Unicode
unicode_string = u'Hola desde España'
print(unicode_string)

# Podemos concatenar dos cadenas utilizando el operador '+'
same_string = 'Hello, ' + 'Bio' + '!'
print(same_string)

# En Python también podemos utilizar wildcards como en la
# función sprintf de C. Por ejemplo:
name = "Guido"
num_emails = 5
print("Hello, %s! You've got %d new emails" % (name, num_emails))
```

```
Hello, Bio!
Hola desde España
Hello, Bio!
Hello, Guido! You've got 5 new emails
```

En el ejemplo anterior, hemos sustituido en el string la cadena *%s* por el contenido de la variable *name*, que es un string, y *%d* por *num_emails*, que es un número entero. Podríamos también utilizar *%f* para números decimales (podríamos indicar la precisión por ejemplo con *%5.3f*, el número tendría un tamaño total de cinco cifras y tres serían para la parte decimal). Hay muchas otras posibilidades, pero deberemos tener en cuenta el tipo de variable que queremos sustituir. Por ejemplo, si utilizamos *%d* y el contenido es string, Python devolverá un mensaje de error. Para evitar esta situación, será recomendado el uso de la función *str()* para convertir el valor a string.

Ahora vamos a presentar otros tipos de datos nativos más complejos: listas, tuplas y diccionarios:

Listas

In [12]:
```
# Definimos una lista con el nombre de los planetas (string)
planets = ['Mercury', 'Venus', 'Earth', 'Mars',
           'Jupiter', 'Saturn', 'Uranus', 'Neptune']
# También puede contener números
prime_numbers = [2, 3, 5, 7]

# Una lista vacía
empty_list = []

# O una mezcla de cualquier tipo:
sandbox = ['3', 'a string', ['a list inside another list', 'second item'], 7.5]
print(sandbox)
```

```
['3', 'a string', ['a list inside another list', 'second item'], 7.5]
```

In [13]:
```
# Podemos añadir elementos a una lista
planets.append('Pluto')
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'
, 'Pluto']
```

In [14]:
```
# O podemos eliminar
planets.remove('Pluto')
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'
]
```

In [15]:
```
# Podemos eliminar cualquier elemento de la lista
planets.remove('Venus')
print(planets)
```

```
['Mercury', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

In [16]:
```
# Siempre que añadamos, será al final de la lista. Una lista está ordenada.
planets.append('Venus')
print(planets)
```

```
['Mercury', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Venus'
]
```

In [17]:
```
# Si queremos ordenarla alfabéticamente, podemos utilizar la función 'sorted()'
print(sorted(planets))
```

```
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus'
]
```

In [18]:
```
# Podemos concatenar dos listas:
monsters = ['Godzilla', 'King Kong']
more_monsters = ['Cthulu']
print(monsters + more_monsters)
```

```
['Godzilla', 'King Kong', 'Cthulu']
```

In [19]:
```
# Podemos concatenar una lista a otra y guardarla en la misma lista:
monsters.extend(more_monsters)
print(monsters)
```

```
['Godzilla', 'King Kong', 'Cthulu']
```

In [20]:
```
# Podemos acceder a un elemento en concreto de la lista:
print(monsters[0])
# El primer elemento de una lista es el 0, por lo tanto, el segundo será el 1:
print(monsters[1])
# Podemos acceder al último elemento mediante números negativos:
print(monsters[-1])
# Penúltimo:
print(monsters[-2])
```

```
Godzilla
King Kong
Cthulu
King Kong
```

In [21]:
```
# También podemos obtener partes de una lista mediante la técnica de 'slicing'
planets = ['Mercury', 'Venus', 'Earth', 'Mars',
           'Jupiter', 'Saturn', 'Uranus', 'Neptune']
# Por ejemplo, los dos primeros elementos:
print(planets[:2])
```

```
['Mercury', 'Venus']
```

In [22]:
```
# O los elementos del segundo al penúltimo:
print(planets[1:-1])
```

```
['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']
```

La técnica de **slicing** es muy importante y nos permite manejar listas de una forma muy sencilla y potente. Será muy importante dominarla para muchos de los problemas que tendremos que resolver en el campo de la bioinformática.

In [23]:
```
# Podemos modificar un elemento en concreto de una lista
monsters = ['Godzilla', 'King Kong', 'Cthulu']
monsters[-1] = 'Kraken'
print(monsters)
```

```
['Godzilla', 'King Kong', 'Kraken']
```

Tuplas

In [24]:
```
# Una tupla es un tipo muy parecido a una lista, pero es immutable, es decir, u
na vez declarada no podemos añadir
# ni eliminar elementos:
birth_year = ('Stephen Hawking', 1942)

# Si ejecutamos la siguiente línea, obtendremos un error de tipo 'TypeError'
birth_year[1] = 1984
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-9de272a7434a> in <module>()
      4
      5 # Si ejecutamos la siguiente línea, obtendremos un error de tipo 'Type
Error'
----> 6 birth_year[1] = 1984

TypeError: 'tuple' object does not support item assignment
```

Los errores en Python suelen ser muy informativos. Una búsqueda en Internet nos ayudará en la gran mayoría de problemas que podamos tener.

In [26]:
```
# Un string también es considerada una lista de caracteres
# Así pues, podemos acceder a una posición determinada (aunque no modificarla):
name = 'Albert Einstein'
print(name[5])

# Podemos separar por el carácter que consideramos un string.
# En este caso, por el espacio en blanco, utilizando la función split()
n, surname = name.split()
print(surname)

# Y podemos convertir un determinado string en una lista de caracteres fácilmen
te:
chars = list(surname)
print(chars)

# Para unir los diferentes elementos de una lista mediante un carácter, podemos
utilizar la función join():
print(''.join(chars))
print('.'.join(chars))
```

```
t
Einstein
['E', 'i', 'n', 's', 't', 'e', 'i', 'n']
Einstein
E.i.n.s.t.e.i.n
```

In [27]:
```
# El operador ',' es el creador de tuplas. Por ejemplo, el típico problema de a
signar el valor de una variable a otra
# en Python puede ser resuelto en una línea de forma muy elegante utilizando tu
plas (se trata de un idiom):
a = 5
b = -5
a,b = b,a
print(a)
print(b)
```

```
-5
5
```

El anterior ejemplo es un *idiom* típico de Python. En la tarcera línea, creamos una tupla (a,b) a la que asignamos los valores uno por uno de la tupla (b,a). Los paréntesis no son necesarios y por eso queda una notación tan reducida.

Diccionarios

Para acabar, presentaremos los diccionarios, una estructura de datos muy útil en la que asignamos un valor a una clave en el diccionario:

In [28]:
```
# Códigos internacionales de algunos países. La clave o 'key' es el código de p
aís y el valor, su nombre:
country_codes = {34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None}

# Podemos buscar
my_code = 34
country = country_codes[my_code]
print(country)
```

```
Spain
```

In [29]:
```
# Podemos obtener todas las claves:
print(country_codes.keys())
```

```
dict_keys([34, 376, 41, 424])
```

In [30]:
```
# O los valores:
print(country_codes.values())
```

```
dict_values(['Spain', 'Andorra', 'Switzerland', None])
```

Es muy importante notar que los valores que obtenemos de las claves o al imprimir un diccionario no están ordenados. Es un error muy común suponer que el diccionario se guarda internamente en el mismo orden en el que fue definido y será una fuente de error habitual no tenerlo en cuenta.

In [33]:
```
# Podemos modificar valores en el diccionario o añadir nuevas claves.

# Definimos un diccionario vacío. country_codes = dict() es una notación equiva
lente:
country_codes = {}

# Añadimos un elmento:
country_codes[34] = 'Spain'

# Añadimos otro:
country_codes[81] = 'Japan'

print(country_codes)
```

```
{34: 'Spain', 81: 'Japan'}
```

In [32]:
```
# Modificamos el diccionario:
country_codes[81] = 'Andorra'

print(country_codes)
```

```
{34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None, 81: 'Andorra'}
```

In [31]:
```
# Podemos asignar el valor vacío a un elemento:
country_codes[81] = None

print(country_codes)
```

```
{34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None, 81: None}
```

Los valores vacíos nos serán útiles para declarar una variable que no sepamos qué valor o qué tipo de valor contendrá y para hacer comparaciones entre variables. Típicamente, los valores vacíos son None o '' en el caso de las cadenas de caracteres.

In [34]:
```
# Podemos asignar el valor de una variable a otra. Es importante que se entiend
an las siguientes líneas:
a = 5
b = 1
print(a, b)
# b contiene la 'dirección' del contenedor al que apunta 'a'
b = a
print(a, b)
```

```
5 1
5 5
```

In [35]:
```
# Veamos ahora qué pasa si modificamos el valor de a o b:
a = 6
print(a, b)
b = 7
print(a, b)
```

```
6 5
6 7
```

Para saber más

Hasta aquí hemos presentado cómo declarar y utilizar variables. Recomendamos la lectura de la documentación oficial *online* para fijar los conocimientos explicados: https://docs.python.org/3/tutorial/introduction.html (https://docs.python.org/3/tutorial/introduction.html)