1 Input/Output y el tipo de dato String

2 Output

Ahora que se han visto los conceptos de **asignación** y **variables**, es indispensable entregar mayor información a los/as usuarios/as al momento de ejecutar un programa. Dicha información puede ser entregada de distintas formas, entre las cuales se pueden mencionar:

- Generación de archivos
- Gráficos
- Base de datos
- Información mediante interfaz gráfica
- Información por pantalla

Considerando que se están entregando contenidos básicos en la programación, se iniciará con la generación de salidas por pantalla. Para ello se utilizará una función nativa llamada `print()`.

Para poder entender la importancia de la función `print()` se utilizará el siguiente ejemplo, donde se pretende calcular el perímetro de una circunferencia de radio 3 cm.

Este código solo consigue que Python entregue solo el resultado numérico, pero no indica ni la unidad de medida ni el cálculo que se realizó (obtención del perímetro).

Lo anterior se solucionará mediante la utilizanción de la función nativa de Python `print()`

[5]:
```
# Ejemplo 1
# Se definen las constantes y variables a utilizar
NUMERO_PI = 3.14
radio = 3

# Se realiza los cálculos respectivos
perimetro_circunferencia = 2 * NUMERO_PI * radio

# Se entrega el resultado
print("El perímetro de la circunferencia, de radio 3 cm, es de",␣
 ↪perimetro_circunferencia, "cm")
```

```
El perímetro de la circunferencia, de radio 3 cm, es de 18.84 cm
```

Como se observa en el Ejemplo 1, la función `print()` permite entregar información detallada, mezclando mensajes (`strings`) y variables (definidas por el/la usuario/a u obtenidas durante la ejecución del programa).

Para realizar lo anterior, dentro de la función `print()` se debe escribir lo que se desea informar al usuario/a, separando los mensajes de las variables con el uso de comas (`,`)

```
print("El perímetro de la circunferencia, de radio 3 cm, es de",perimetro_circunferencia,"cm")
```

Del ejemplo anterior se puede observar que la salida generada tiene mayor información para el/la usuario/a.

La función `print()` permite entregar información detallada de los cálculos realizados y, además, permite entregar más de una salida.

Utilizando el ejemplo anterior, se calculará el perímetro y el área de una circunferencia.

[6]:
```
# Ejemplo 2
# Se definen las constantes y variables a utilizar
NUMERO_PI = 3.14
radio = 3

# Se realiza los cálculos respectivos
perimetro_circunferencia = 2 *NUMERO_PI * radio
area_circunferencia = NUMERO_PI * radio**2

# Se entrega el resultado
print("El perímetro de la circunferencia, de radio 3 cm, es de",␣
 ↪perimetro_circunferencia, "cm")
print("El área de la circunferencia, de radio 3 cm, es de", area_circunferencia,␣
 ↪"cm2")
```

```
El perímetro de la circunferencia, de radio 3 cm, es de 18.84 cm
El área de la circunferencia, de radio 3 cm, es de 28.26 cm2
```

Como se aprecia en el Ejemplo 2, cada vez que se utiliza la función `print()`, se realiza un salto de línea (`\n`). Esto se debe a que esta función fue creada para que realice esa acción, a menos que se indique otra cosa.

Para cambiar la opción anterior y modificar la acción que realiza la función `print()` al ser ejecutada es la siguiente:

```
print(<tipo de dato o mensaje a imprimir>, end = " ")
```

[7]:
```
# Ejemplo 3
# Se definen las constantes y variables a utilizar
NUMERO_PI = 3.14
radio = 3

# Se realiza los cálculos respectivos
perimetro_circunferencia = 2 * NUMERO_PI * radio
area_circunferencia = NUMERO_PI * radio**2

# Se entrega el resultado
print("El perímetro de la circunferencia, de radio 3 cm, es de",␣
 ↪perimetro_circunferencia, "cm y", end = " ")
print("el área de la circunferencia, de radio 3 cm, es de", area_circunferencia,␣
 ↪"cm2")
```

```
El perímetro de la circunferencia, de radio 3 cm, es de 18.84 cm y el área de la
circunferencia, de radio 3 cm, es de 28.26 cm2
```

3 Input

Con lo visto anteriormente, ya se puede entregar información por pantalla, pero los programas son monótonos y de seguir trabajando de esa manera, se tendría que hacer un programa para cada caso.

Por ejemplo, se tendría que hacer un programa para determinar el área y perímetro de una circunferencia para cada valor del radio.

Es por ello que los lenguajes de programación tienen la posibilidad de alimentar los programas con información. Esto se puede realizar principalmente mediante las siguientes posibilidades:

- Lectura de archivos
- Ingreso de información por teclado

Considerando que se están entregando contenidos básicos en la programación, se iniciará con la alimentación de información mediante el ingreso por teclado. Para ello, se utilizará una función nativa llamada `input()`.

La función `input()` detiene la ejecución del programa y espera que el/la usuario/a ingrese datos por teclado y presione la tecla **enter** para señalar que terminó de ingresar. Posteriormente, la función captura los datos ingresados por el usuario y los retorna como una cadena de texto (`string`), una vez entregado el texto, el programa sigue su ejecución.

Para poder comprender el modo de operación de la función `input()`, se utilizará el Ejemplo 4, donde se solicita obtener el cálculo del perímetro y área de un rectángulo. En particular para este caso, se desea que el usuario pueda indicar los el valor de los lados, cada vez que ejecute el programa.

[8]:
```
# Ejemplo 4
# Se solicita la información al usuaio/a y se guarda en variables
lado_a = input()
lado_b = input()

# Se realizan los cálculos respectivos
perimetro_rectangulo = 2*lado_a + 2*lado_b
area_rectangulo = lado_a*lado_b

# Se entrega el resultado
print("El perímetro y área del rectángulo, de lados", lado_a, "cm y ", lado_b,␣
 ↪"cm es",
      perimetro_rectangulo,"cm y",area_rectangulo,"cm2 respectivamente")
```

```
3
4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-565ac7a41754> in <module>
      6 # Se realizan los cálculos respectivos
      7 perimetro_rectangulo = 2*lado_a + 2*lado_b
----> 8 area_rectangulo = lado_a*lado_b
      9
```

```
    10 # Se entrega el resultado

TypeError: can't multiply sequence by non-int of type 'str'
```

El programa anterior entrega un mensaje de error producto de que las entradas solicitadas al usuario/a fueron solicitadas con la función `input()`, la cual genera un tipo de dato `string`.

Para poder realizar el cálculo del área y perímetro de manera correcta, es necesario realizar un cambio de *tipo de dato*:

string –> número

En este caso en particular, se utilizará la función `float()`, pues esta nos permite capturar tanto valores enteros como no enteros (Ejemplo 5).

[9]:
```
# Ejemplo 5
# Se solicita la información al usuario/a y se guarda en variables
lado_a = input()
lado_b = input()

# Se realiza el cambio de tipo de dato respectivo
lado_a = float(lado_a)
lado_b = float(lado_b)

# Se realizan los cálculos respectivos
perimetro_rectangulo = 2*lado_a + 2*lado_b
area_rectangulo = lado_a*lado_b

# Se entrega el resultado
print("El perímetro y área del rectángulo, de lados", lado_a, "cm y ", lado_b,␣
 ↪"cm es",
      perimetro_rectangulo, "cm y", area_rectangulo, "cm2 respectivamente")
```

```
3
4
El perímetro y área del rectángulo, de lados 3.0 cm y  4.0 cm es 14.0 cm y 12.0
cm2 respectivamente
```

Otra característica que posee la función `input()` es que permite agregar información (entrada) al usuario/a para que pueda saber con exactitud lo que se está solicitando. Para ello, se debe escribir el mensaje que se desea entregar al usuario/a dentro del paréntesis como cadena de texto (entre comillas).

Básicamente podemos entregarle cualquier string a la función `input()` y este será mostrado al usuario al momento de pedirle que ingrese los datos (Ejemplo 6).

[10]:
```
# Ejemplo 6
# Se solicita la información al usuaio/a y se guarda en variables
lado_a = input("Favor ingresar la medida del primer lado del rectángulo: ")
lado_b = input("Favor ingresar la medida del segundo lado del rectángulo: ")
```

```
# Se realiza el cambio de tipo de dato respectivo
lado_a = float(lado_a)
lado_b = float(lado_b)

# Se realizan los cálculos respectivos
perimetro_rectangulo = 2*lado_a + 2*lado_b
area_rectangulo = lado_a*lado_b

# Se entrega el resultado
print("El perímetro y área del rectángulo, de lados", lado_a, "cm y ", lado_b,␣
 ↪"cm es",
      perimetro_rectangulo, "cm y", area_rectangulo, "cm2 respectivamente")
```

```
Favor ingresar la medida del primer lado del rectángulo: 3
Favor ingresar la medida del segundo lado del rectángulo: 4
El perímetro y área del rectángulo, de lados 3.0 cm y  4.0 cm es 14.0 cm y 12.0
cm2 respectivamente
```

Existe la posibilidad de poder simplificar el código anterior utilizando **funciones anidadas**. Esto se traduce en llamar una función dentro de otra

```
lado_a = float(input("Favor ingresar la medida del primer lado del rectángulo: "))
```

Realizando el cambio en el código (Ejemplo 7), este queda de la siguiente manera:

[11]:
```
# Ejemplo 7
# Se solicita la información al usuaio/a, se realiza el cambio de tipo de dato y␣
 ↪se guarda en variables
lado_a = float(input("Favor ingresar la medida del primer lado del rectángulo:␣
 ↪"))
lado_b = float(input("Favor ingresar la medida del segundo lado del rectángulo:␣
 ↪"))

# Se realizan los cálculos respectivos
perimetro_rectangulo = 2*lado_a + 2*lado_b
area_rectangulo = lado_a*lado_b

# Se entrega el resultado
print("El perímetro y área del rectángulo, de lados", lado_a, "cm y ", lado_b,␣
 ↪"cm es",
      perimetro_rectangulo, "cm y", area_rectangulo, "cm2 respectivamente")
```

```
Favor ingresar la medida del primer lado del rectángulo: 3
Favor ingresar la medida del segundo lado del rectángulo: 4
El perímetro y área del rectángulo, de lados 3.0 cm y  4.0 cm es 14.0 cm y 12.0
cm2 respectivamente
```

NOTA: Pese a que la segunda versión del código posee menos líneas que el código anterior, se recomienda no abusar de las funciones anidadas. Como regla para este curso, siempre se recomienda

privilegiar la *legibilidad* (es decir, que el código sea entendible), por sobre la *reducción* de las líneas del código.

4 Abstracción de procesos

Una forma de ayudarnos a entender como organizar un programa es entendiendo que este básicamente solo es un proceso de transformación de entradas en una (o varias) salida en particular. La forma más sencilla de enfrentarse a un problema de programación y entender qué es lo que debo hacer, es organizar el código en 3 bloques: * Entrada: Correspondiente a los valores que me entregarán para operar. * Procesamiento: Correspondiente a las operaciones que debo hacer para generar la(s) salida(s). * Salida: Donde informo al usuario del resultado del proceso.

Una formá básica de estructurarlo, considerando además la posibilidad de valores definidos previamente, es la siguiente:

```
# CONSTANTES

# ENTRADAS

# PROCESAMIENTO

# SALIDA
```

Revisemos como esto organizaría el ejemplo anterior:

[12]:
```
# ENTRADAS
# Se solicita la información al usuaio/a, se realiza el cambio de tipo de dato y␣
 ↪se guarda en variables
lado_a = float(input("Favor ingresar la medida del primer lado del rectángulo:␣
 ↪"))
lado_b = float(input("Favor ingresar la medida del segundo lado del rectángulo:␣
 ↪"))

# PROCESAMIENTO
# Se realizan los cálculos respectivos
perimetro_rectangulo = 2*lado_a + 2*lado_b
area_rectangulo = lado_a*lado_b

# SALIDA
# Se entrega el resultado
print("El perímetro y área del rectángulo, de lados", lado_a, "cm y ", lado_b,␣
 ↪"cm es",
      perimetro_rectangulo, "cm y", area_rectangulo, "cm2 respectivamente")
```

```
Favor ingresar la medida del primer lado del rectángulo: 10
Favor ingresar la medida del segundo lado del rectángulo: 30
El perímetro y área del rectángulo, de lados 10.0 cm y  30.0 cm es 80.0 cm y
300.0 cm2 respectivamente
```

En este caso, como no tenemos constantes, tenemos solo 3 bloques, las instrucciones donde se piden los lados (entradas), el proceso de cálculo de perímetro y área (procesamiento) y la sección donde se informa al usuario del resultado (salida).

Uno de los principales problemas que se tienen al momento de querer plantear una solución en un lenguaje de programación es que es muy complejo pensar simultáneamente en la solución de un problema y en cómo plantearla en Python (u otro lenguaje). Para facilitar esa tarea, se utiliza un razonamiento estructurado llamado **abstracción de procesos**, la cual consta de una serie de pasos:

1. Identificar las entradas del problema
2. Identificar las transformaciones que el programa debe hacer usando los datos de entrada
3. Identificar las salidas a entregar, que son el resultado de las transformaciones anteriores.

La abstracción es la habilidad **más importante** al momento de programar, e incluso, de resolver cualquier problema complejo. Es la herramienta que permite responder a las preguntas claves que están presentes al momento de proponer una solución:

1. ¿**Qué** se quiere hacer? - corresponde a saber cuál es el problema
2. ¿**Con qué** se cuenta? - apunta a conocer las entradas del problema
3. ¿**Cómo** se hace? - se centra en saber qué hay que hacer con las entradas para lograr el objetivo
4. ¿**Para qué** se hace? - corresponde a identificar el resultado que se quiere obtener, es decir, las salidas que entraga la solución

Si se domina correctamente esta técnica, serán capaces de aplicarla incluso para problemas muy complejos, descomponiendo un enunciado en fragmentos más pequeños. Se puede resolver cada uno de los fragmentos o sub-problemas utilizando soluciones ya conocidas y luego combinar las soluciones parciales para resolver el problema completo.

La única forma de desarrollar esta habilidad es ejercitándola constantemente y desafiándose a uno mismo a ser creativo e ingenioso.

Es importante tener en cuenta que, en ocasiones, se tendrán que usar los conocimientos de otras áreas para poder reunir toda la información que se necesite para resolver un problema.

4.1 Ejemplo de aplicación

Considerando que ya se han visto los conceptos básicos para realizar un programa en Python, se plantea un enunciado para poder practicar lo aprendido hasta el momento:

Un tren parte desde la estación Los Domínicos en dirección hacia San Pablo, situada a 17.7 km de distancia, con una rapidez promedio de 60 km/h. Simultáneamente, otro tren parte en sentido contrario desde la estación San Pablo con una rapidez media de 75 km/h. ¿A qué distancia de la estación Los Domínicos se cruzan ambos trenes y a cuántos minutos de la partida?

Lo primero es realizar la **abstracción de procesos**:

Entradas

- Distancia entre ambos puntos (`d`): 17.7 km
- Rapidez del primer tren (`v1`): 60 km/h
- Rapidez del segundo tren (`v2`): 75 km/h

Procesamiento: Por física se sabe que:

$$d = v * t$$

Además, se sabe que la distancia total se obtiene como:

$$d = v1 * t + v2 * t = (v1 + v2) * t$$

De lo anterior, se puede conocer el timepo (en horas) que tarda los trenes en encontrarse:

$$t = d / (v1 + v2)$$

Se sabe que una hora tiene 60 minutos, por lo que el tiempo en minutos está dado por:

$$tm = t * 60$$

Finalmente, se puede determinar a qué distancia de la estación Los Domínicos se cruzan los trenes de la siguiente manera:

$$dc = v1 * t$$

Salidas:

- Distancia de la estación Los Domínicos en que se cruzan los trenes (`dc`)
- Tiempo, en minutos, transcurrido desde la partida hasta el cruce (`tm`)

En el ejercicio anterior se utilizó la abstracción de procesos para obtener la idea de solución de un problema. El siguiente paso es tomar esa idea y convertirla en un programa ordenado y fácil de leer. Para eso se utilizará lo que se ha aprendido de Python hasta ahora e incorporar las buenas prácticas de programación.

La transformación de la idea en un programa consta de los siguientes pasos:

1. Escribir las entradas que se identificaron como variables, con nombres representativos.
2. Escribir las transformaciones identificadas como sentencias de Python, almacenando los valores intermedios en nuevas variables
3. Mostrar las salidas identificadas, al usaurio/a, utilizando la función `print()` y un mensaje de texto que indique al usuario/a qué es la información que está recibiendo.

Dicho lo anterior, se tiene el siguiente código:

[13]:
```
# ENTRADAS
distancia_total = 17.7
rapidez_tren_1 = 60
rapidez_tren_2 = 75

# PROCESAMIENTO
tiempo_horas = distancia_total / (rapidez_tren_1 + rapidez_tren_2)
tiempo_minutos = tiempo_horas * 60
distancia_cruce = rapidez_tren_1 * tiempo_horas

# SALIDA
print("Los trenes se cruzan a", distancia_cruce, "km de la estación Los␣
 ↪Domínicos a",
      tiempo_minutos, "minutos de la partida")
```

```
Los trenes se cruzan a 7.866666666666665 km de la estación Los Domínicos a
7.866666666666665 minutos de la partida
```

Si bien el programa resuelve correctamente el problema, hay algunas cosas que podrían no quedar claras a otras personas que lean el programa. Por ello, siempre es importante ayudar a un posible lector con un poco de información adicional que le ayude a entender la estructura del programa y las tareas que realiza. La herramienta que se tiene para este fin son los comentarios, líneas del programa escritas y que no son tomadas en cuenta por el interprete de Python al momento de ejecutar el programa.

Para escribir un comentario, basta con incluir el símbolo gato/numeral/almohadilla (#) al comienzo de una línea. Con eso, el intérprete omitirá la línea completa, por lo que no se debe incluir sentencias que se ejecuten en una línea que tenga un comentario.

También existe formas de escribir comentarios que abarquen varías líneas a la vez, esto se logra utilizando triple comillas, ya sean comillas simples o dobles.

```
"""
esto es un ejemplo de un
comentario multilínea
"""
```

Otro aspecto importante es que se escriben con mayúsculas aquellos comentarios que identifiquen distintas secciones de un programa, mientras que los que explican lo que hace el programa se escriben con minúscula.

El desarrollo del ejercicio anterior quedaría de la siguiente manera:

[14]:
```
# ENTRADAS
distancia_total = 17.7
rapidez_tren_1 = 60
rapidez_tren_2 = 75

# PROCESAMIENTO
# Se calcula el tiempo total, en horas
tiempo_horas = distancia_total/(rapidez_tren_1 + rapidez_tren_2)

# Se calcula el tiempo en minutos
tiempo_minutos = tiempo_horas * 60

# Se calcula la distancia de cruce
distancia_cruce = rapidez_tren_1 * tiempo_horas

# SALIDA
# Se entrega la distancia (en km) desde la estación Los Domínicos a la que se␣
 ↪cruzan los trenes
# y el tiempo (en minutos) transcurridos desde su partida
print("Los trenes se cruzan a", distancia_cruce, "km de la estación Los␣
 ↪Domínicos a",
      tiempo_minutos,"minutos de la partida")
```

```
Los trenes se cruzan a 7.866666666666665 km de la estación Los Domínicos a
7.866666666666665 minutos de la partida
```

5 String

Los **strings** son un tipo de dato que permite almacenar textos. En particular, a diferencia de los tipos de datos numéricos, el string es un tipo de dato *compuesto*, es decir, se puede descomponer en partes.

Para que el programa entienda algo como string, es necesario definirlos con comillas, ya sean 'simples' o "dobles", siempre siendo consistentes entre las comillas de apertura y las de cierre.

[15]:
```
# Ejemplo 8: Se definirá un string con comillas simples y luego con comillas␣
↪dobles
print('hola mundo')
print("Hoy hay que estudiar")
```

```
hola mundo
Hoy hay que estudiar
```

Lo que se ilustró en el **Ejemplo 8** fue la impresión por pantalla de dos strings definidos, tanto con comillas simples, como con comillas dobles.

Si se quiere definir un **string** mezclando ambas comillas, arrojará un mensaje de error como lo muestran los **Ejemplos 9** y **10**:

[16]:
```
# Ejemplo 9: Se definirá un string que inicialmente comienza con comillas␣
↪simples y
# termina con comillas dobles
print('hola mundo")
```

```
  File "<ipython-input-16-acf7a5bc1790>", line 3
    print('hola mundo")
                       ^
SyntaxError: EOL while scanning string literal
```

[17]:
```
# Ejemplo 10: Se definirá un string que inicialmente comienza con comillas␣
↪dobles y
# termina con comillas simples
print("hola mundo')
```

```
  File "<ipython-input-17-ae72b7e03376>", line 3
    print("hola mundo')
                       ^
SyntaxError: EOL while scanning string literal
```

Además de las comillas simples y dobles, Python ofrece otra forma de poder definir `string` de gran tamaño, llamados `docstring`. El uso que se le da a estas cadenas de texto son, principalmente, para documentar secciones del código.

Este tipo de string se define con triple comillas.

El uso de un `docstring`se puede ver en el **Ejemplo 11**:

[18]:
```
# Ejemplo 11: Creación de un string multilínea o docstring
texto = """La idea de este tipo de strings es generar
documentación para que otros usuarios pueden entender
lo que se está haciendo en este código"""

print(texto)
```

```
La idea de este tipo de strings es generar
documentación para que otros usuarios pueden entender
lo que se está haciendo en este código
```

5.1 Uso de Variables

Los `strings`, al igual que los otros tipos de datos vistos se pueden asignar a variables para su posterior uso.

Para lograr esto, se define una variable a la que será asignado un determinado string (Ejemplo 12):

[19]:
```
# Ejemplo 12: se definirá un string en una variable para, posteriormente, ser␣
↪mostrada por pantalla
texto = "hola mundo"

print(texto)
```

```
hola mundo
```

Es importante recordar que, **por convención**, las variables creadas deben estar escritas en **minúsculas** y, si son varias palabras, se deben separar con **guiones bajos**:

[20]:
```
# Ejemplo 13: definición de un string utilizando buenas prácticas

mensaje_correcto = "hola mundo"

mensajeMalEscrito = "hola mundo_2"

print(mensaje_correcto)
print(mensajeMalEscrito)
```

```
hola mundo
hola mundo_2
```

Como se aprecia en el **Ejemplo 13**, las buenas prácticas no afecta el funcionamiento del código, no obstante, el código no solo debe ser entendible por Python, sino también por otros programadores. Por eso es necesario respetar las convenciones de nombres.

Python permite hacer uso de comillas dentro de un `string` si estas difieren con las utilizadas para definirlo. Lo anterior se refiere a que, si el string fue definido con comillas dobles, el texto que se quiere resaltar debe estar con comillas simples y viceversa.

[21]:
```
# Ejemplo 14: Imprimir mensajes que incorporen comillas
mensaje_1 = "Este es un cuadernillo de 'Google Colab'"
mensaje_2 = "Se imprime un ejemplo para 'comprender mejor'"

print(mensaje_1)
print(mensaje_2)
```

```
Este es un cuadernillo de 'Google Colab'
Se imprime un ejemplo para 'comprender mejor'
```

Del mismo modo, si quisiera usar las mismas comillas dentro del string y para definirlo, puedo hacerlo usando el caracter de escape o backslash (\), como se presenta en el Ejemplo 15.

[22]:
```
# Ejemplo 15: Imprimir mensajes que incorporen comillas
mensaje_1 = "Este es un cuadernillo de \"Google Colab\""
mensaje_2 = 'Se imprime un ejemplo para \'comprender mejor\''

print(mensaje_1)
print(mensaje_2)
```

```
Este es un cuadernillo de "Google Colab"
Se imprime un ejemplo para 'comprender mejor'
```

5.2 Operadores y Operaciones sobre Strings

Al igual que con los tipos de datos numéricos, en Python también existen **operadores** para trabajar con strings.

Estos operadores son utilizados la realizar cálculos matemáticos sencillos y sus símbolos son `+` y `*`, los cuales representan a la suma y multiplicación de números respectivamente.

Sin embargo,cuándo estos se usan para los strings, pasan a realizar operaciones de concatenación (pegar strings) y repetición (crear copias del string) respectivamente.

5.2.1 Concatenación

Esta operación consiste en **unir** dos o más cadenas de `strings` para formar una nueva cadena

`string_1 + string_2 -> string_1string_2`

El **Ejemplo 16** muestra lo indicado anteriormente:

[23]:
```
# Ejemplo 16: concatenación de dos strings

texto_1 = "hola"
texto_2 = "mundo"

nuevo_texto = texto_1 + texto_2
```

```
nuevo_texto_2 = texto_1 + " " + texto_2

print(nuevo_texto)
print(nuevo_texto_2)
```

```
holamundo
hola mundo
```

Vale la pena destacar que la concatenación funciona únicamente si **ambos elementos** son del tipo string. Si intento unir con otro tipo de dato, como por ejemplo `int` o `float`, Python entregará un error.

El que será distinto según el tipo de dato del primer operador de la operación (Ejemplos 17 y 18)

[24]:
```
# Ejemplo 17
'hola' + 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-19895f43b8c3> in <module>
      1 # Ejemplo 17
----> 2 'hola' + 3

TypeError: can only concatenate str (not "int") to str
```

[25]:
```
# Ejemplo 18
3 + 'hola'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-7c12ea9d9fb7> in <module>
      1 # Ejemplo 18
----> 2 3 + 'hola'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

5.2.2 Repetición

Esta operación se realiza teniendo como primer operador un string y como segundo operador un número entero. Consiste en **repetir** un `string` tantas veces como indique el número al cuál este siendo multiplicado:

`string_1 * 3 -> string_1string_1string_1`

El **Ejemplo 19** muestra lo indicado anteriormente:

[26]:
```
# Ejemplo 19: repetición de un string

```

```
texto_1 = "hola"

nuevo_texto = texto_1 * 3
nuevo_texto_2 = 3 * texto_1

print(nuevo_texto)
print(nuevo_texto_2)
```

```
holaholahola
holaholahola
```

Adicionalmente, Python ofrece diversas funciones nativas que se pueden realizar con el tipo de dato `string`. Entre ellas vale destacar: * `len(<string>)`: devuelve la cantidad de caracteres del string * `str(<expresion>)`: devuelve la expresión ingresada en formato string

El **Ejemplo 20** ilustra lo mencionado anteriormente:

[27]:
```
# Ejemplo 20: uso de funciones de strings

oracion = "este es un ejemplo para calcular el largo"

largo = len(oracion)

numero = 66
texto_numero = str(numero)

numero_nuevo = numero*2
repeticion_numero = texto_numero*2

print(largo)
print(numero_nuevo)
print(repeticion_numero)
```

```
41
132
6666
```

En el ejemplo anterior se calculó el largo o la cantidad de caracteres que componían el string `oracion`.

Posterior a ello, se transformó el número 66 a `string` y se realizó una comparación entre dos operaciones: multiplicación del número 66 por 2 y la multiplicación del número transformado a strings por 2 (repetición).

5.3 Indexación

Del mismo modo, existe una manera de acceder a los caracteres contenidos en el `string` de manera **individual**.

Para lograr lo anterior, se tiene que llamar a la variable que contiene el `string` y acompañarlo de corchetes (`[ ]`) e indicar, dentro de el, la posición numérica (contando de izquierda a derecha) de

dicho caracter.

Es importante tener en cuenta que el primer caracter siempre estará en la posición `0` y así sucesivamente.

Tampoco olvidar que los espacios **también son considerados** al momento de recorrer un `string`.

[28]:
```
# Ejemplo 21: Encontrar un caracter mediante su posición
palabra = "Hola Mundo"

# Se quiere mostar por pantalla la letra "M", por lo que tendremos que indicar␣
 ↪su posición
caracter = palabra[5]

print(caracter)
```

```
M
```

IMPORTANTE: Nunca se debe olvidar que el primer caracter se encuentra en la posición **0** y el último caracter se encuentra en la posición **largo - 1** o, en otras palabras, `len(<string>) - 1`.

Dicho esto, se puede acceder a cualquier posición existente dentro del 0 y el largo - 1. Si se accede a una posición fuera de esos rangos Python entrega el error `IndexError: string index out of range` el cuál es uno de los más comunes que comenten los programadores novatos.

[29]:
```
# Ejemplo 22: encontrar un caracter mediante su posición

palabra = "Hola Mundo"

# Se quiere mostar por pantalla la letra "M", por lo que tendremos que indicar␣
 ↪su posición
caracter = palabra[25]

print(caracter)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-29-9631bd5e2698> in <module>
      4
      5 # Se quiere mostar por pantalla la letra "M", por lo que tendremos que␣
 ↪indicar su posición
----> 6 caracter = palabra[25]
      7
      8 print(caracter)

IndexError: string index out of range
```

Además de acceder a posiciones determinadas, es posible acceder a determinadas zonas o partes de un `string`. Para ello, existe el operador *slice* (pues "corta" parte del string)

Este operador, utiliza la misma sintaxis de corchetes de la indexación, pero recibe dos parámetros separados por el cáracter dos puntos `:`.

`string[x:y]`

- **x**: corresponde a la primera posición que se quiere acceder.
- **y**: corresponde a la posición posterior a la última que se quiere acceder (esta posición no es guardada).

[30]:
```
# Ejemplo 23: encontrar una cadena de caracteres mediante su posición

palabra = "Hola Mundo"

# Se accede a las posiciones 2 a 7 (no incluye la última)
cadena = palabra[2:8]

print(cadena)
```

```
la Mun
```

5.4 Métodos

Los strings,como todo tipo de dato en Python son **objetos**, que, en palabras simples, significa que poseen características y acciones propias que ningún otro tipo de dato puede realizar. Estas acciones se denominan **métodos** y realizan operaciones que para otros tipos de datos no tienen sentido.

Por ejemplo, nos podría interesar cambiar un string de mayúsculas a minúsculas, pero eso no tiene mucho sentido sobre un `int`, `float` o `boolean`.

Algunos métodos interesantes de los string que podemos usar son:

- `<string>.lower()`: retorna una copia del string donde todas sus letras son minúsculas.
- `<string>.upper()`: retorna una copia del string donde todas sus letras son mayúsculas.
- `<string>.title()`: retorna una copia del string donde la primera letra de cada palabra de un string está en mayúscula.
- `<string>.swapcase()`: retorna una copia del string donde cada minúscula es convertida a mayúscula y viceversa.
- `<string>.islower()`: retorna `True` si los caracteres del strings son solo letras minúsculas.
- `<string>.isupper()`: retorna `True` si los caracteres del strings son solo letras mayúsculas.
- `<string>.isdigit()`: retorna `True` si los caracteres del strings son solo numéricos.
- `<string>.isalpha()`: retorna `True` si los caracteres del strings son solo letras.

MANOS A LA OBRA: * Revisa que ocurre con `.isdigit()` si el string representa un número negativo, como por ejemplo `-32`, o un número flotante como `3.1456`. * Revisa que ocurre con `.isalpha()` si el string tiene tildes.

[31]:
```
# Ejemplo 24: método .lower() y .upper()

palabra = "Hola Mundo"

palabra_minuscula = palabra.lower()
```

```
palabra_mayuscula = palabra.upper()

print(palabra_minuscula)
print(palabra_mayuscula)
print(palabra)
```

```
hola mundo
HOLA MUNDO
Hola Mundo
```

Vale la pena notar que al usar métodos en los string, estos normalmente entregan **copias** de este con la transformación solicitada, sin embargo, el string original no es cambiado por estos métodos.

Del Ejemplo 24 podemos ver que tenemos tres variables, cada una con una copia distinta del string. Del mismo modo, el Ejemplo 25 ilustra como podríamos usar la capacidad de las variables de ser reutilizadas, para aplicar transformaciones sucesivas a un string.

[32]:
```
# Ejemplo 25: método .title() y .swapcase()

palabra = "hola mundo"

palabra_titulo = palabra.title()
palabra_cambiada = palabra_titulo.swapcase()

print(palabra_titulo)
print(palabra_cambiada)
```

```
Hola Mundo
hOLA mUNDO
```

[33]:
```
# Ejemplo 26: Método .islower() e isupper()

palabra = "Hola Mundo"
palabra_2 = "HOLA MUNDO"
palabra_3 = "hola mundo"

palabra_minuscula = palabra.islower()
palabra_minuscula_2 = palabra_3.islower()

palabra_mayuscula = palabra.isupper()
palabra_mayuscula_2 = palabra_2.isupper()

print(palabra_minuscula)
print(palabra_minuscula_2)

print(palabra_mayuscula)
print(palabra_mayuscula_2)
```

```
False
True
```

```
False
True
```

[34]:
```python
#Ejemplo 27: Método is.alpha() y is.digit()

palabra = "HolaMundo"
palabra_2 = "espero que pronto sea 31 de diciembre"
palabra_3 = "15842"

palabra_letra = palabra.isalpha()
palabra_letra_2 = palabra_2.isalpha()

palabra_numero = palabra_2.isdigit()
palabra_numero_2 = palabra_3.isdigit()

print(palabra_letra)
print(palabra_letra_2)

print(palabra_numero)
print(palabra_numero_2)
```

```
True
False
False
True
```

6 Bibliografía

6.1 Abstracción

Beecher, K. (2017). Problem-Solving and Decomposition. En *Computational Thinking* (p. 39). BCS.

6.2 Comentarios

Beecher, K. (2017). Tutorial for Python Beginners. En *Computational Thinking* (p. 109). BCS.

GeeksforGeeks. (2022, 11 abril). *Python Comments*. Recuperado 2 de agosto de 2022, de https://www.geeksforgeeks.org/python-comments/

Van Rossum, G., Warsaw, B., & Coghlan, N. (2001, 21 julio). *PEP 8 – Style Guide for Python Code*. Python Enhancement Proposals. Recuperado 1 de agosto de 2022, de https://peps.python.org/pep-0008/

Shaw, Z. (2017). Comments and Pound Characters. En *Learn Python 3 the Hard Way* (p. 14). Addison-Wesley.

6.3 Funciones nativas de Python

GeeksforGeeks. (2021, octubre 29). *Output using print() function*. Recuperado 2 de agosto de 2022, de https://www.geeksforgeeks.org/python-output-using-print-function/

Python Software Foundation. (2022, agosto 2). *Built-in Functions*. Python Documentation. Recuperado 2 de agosto de 2022, de https://docs.python.org/3/library/functions.html

Shaw, Z. (2017). Prompting People. En *Learn Python 3 the Hard Way* (p. 40). Addison-Wesley.

6.4 Formato y buenas prácticas

Van Rossum, G., Warsaw, B., & Coghlan, N. (2001, 21 julio). *PEP 8 – Style Guide for Python Code*. Python Enhancement Proposals. Recuperado 1 de agosto de 2022, de https://peps.python.org/pep-0008/

6.5 Strings

GeeksforGeeks. (2022, 28 julio). Python String. Recuperado 4 de agosto de 2022, de https://www.geeksforgeeks.org/python-string/

Python Software Foundation. (2022a, agosto 4). *An Informal Introduction to Python*. Python 3.10.6 documentation. Recuperado 4 de agosto de 2022, de https://docs.python.org/3/tutorial/introduction.html#strings

Python Software Foundation. (2022b, agosto 4). *Built-in Types*. Python 3.10.6 documentation. Recuperado 4 de agosto de 2022, de https://docs.python.org/3/library/stdtypes.html#textseq