Referencia Rápida de Python

FPI/FCyP

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SANTIAGO DE CHILE

Tipos Básicos

`int`	Números enteros	`42`
`float`	Números flotantes	`1.618`
`bool`	Valores lógicos	`True`
`str`	Strings	`"Monty"`
`list`	Listas	`[1, 2, 3]`

Operaciones Básicas

`input("Mensaje")`	Muestra `Mensaje` y recibe entrada por teclado del usuario
`print(valor,...)`	Muestra por pantalla los `valores` separados por coma
`x = y`	Asignación, `x` toma el valor de la expresión `y`

Operadores Aritméticos

A igual nivel de precedencia (P) en las operaciones, se agrupan de a pares de izquierda a derecha.

Ejemplo	Operación	P	T[a]
`x ** y`	Exponenciación[b]	1	B
`+ x`	Identidad	2	U
`- x`	Cambio de signo	2	U
`x * y`	Multiplicación	3	B
`x / y`	División	3	B
`x // y`	División entera	3	B
`x % y`	Módulo (resto[c])	3	B
`x + y`	Suma	4	B
`x - y`	Resta	4	B

Los operadores unarios (U) toman solo un operando.
Los operadores binarios (B) toman dos.
Todos los operadores aritméticos binarios tienen una versión de asignación como `+=`, que se usa como en el ejemplo:

`x += y` se interpreta como `x = x + y`

[a] Tipo
[b] Se agrupan de derecha a izquierda.
[c] De la división entera.

Operadores lógicos

A igual nivel de precedencia en las operaciones, se agrupan de a pares de izquierda a derecha.

Ejemplo	Operación	P	T
`x > y`	Mayor que	5	B
`x >= y`	Mayor o igual que	5	B
`x < y`	Menor que	5	B
`x <= y`	Menor o igual que	5	B
`x == y`	Igual que	5	B
`x != y`	Distinto que	5	B
`x in l`	Pertenece a	5	B
`x not in l`	No pertenece a	5	B
`not p`	Negación	6	U
`p and q`	Y lógico	7	B
`p or q`	O lógico	8	B

Decisiones con `if`

Ejecutar sentencias que solo deben ocurrir al cumplirse la `<condición>` (expresión booleana):

```
if <condición>:
    <sentencias condicionadas>
```

Ejecutar sentencias que solo deben ocurrir al cumplirse una condición, y otras en caso que no:

```
if <condición>:
    <sentencias condicionadas>
else:
    <sentencias alternativas>
```

Ejecutar sentencias que solo deben ocurrir al cumplirse una condición, otras en caso de que no se cumpla la primera condición, pero si una segunda condición[a], y otras en caso de que no se cumpla la primera ni la segunda:

```
if <condición 1>:
    <sentencias condicionadas 1>
elif <condición 2>:
    <sentencias condicionadas 2>
else:
    <sentencias alternativas>
```

[a] Tantas condiciones secundarias (`elif`) como se necesite.

Bloques de decisiones

Cada secuencia `if`, `if-else` o `if-elif-else` es un bloque independiente.
Dentro de un mismo bloque se ejecutarán las sentencias condicionadas a la primera condición válida comprobada.
Si existieran 2 bloques de decisión consecutivos se ejecutarán las sentencias condicionadas a la primera condición válida para cada bloque de manera independiente.

Ciclos con `while`

Realizar una repetición condicionada de sentencias:

```
while <condición>:
    <sentencias_a_repetir>
```

Donde `condición` es una expresión booleana y `<sentencias_a_repetir>` (última instrucción indentada con respecto a `while`) es la secuencia de instrucciones a repetir.
A través de este bloque, se asegura la ejecución de las `<sentencias_a_repetir>` mientras se cumpla la `condición` señalada.
Se puede repetir cualquier sentencia, incluyendo otros ciclos.

Banderas (*Flags*)

Parte de la `<condición>` puede ser una variable booleana: esta indica si se debe continuar o no, según qué ocurra en el ciclo:

```
i = 1
keep_going = True
while keep_going and i <= 5:
    if i % 2 == 0:
        keep_going = False
    i += 1
```

Haciendo uso de decisiones (`if`) y de una bandera (`keep_going`), se puede verificar si se cumple una condición adicional bajo la cual detener el ciclo antes de que el iterador `i` llegue a su límite.

Listas

`lista = [1, 2, 3]`	Define un objeto de tipo lista.
`lista[i]`	Retorna el elemento en la posición `i`. Soporta de `0` a `n-1`[a] de izquierda a derecha y de `-1` a `-n` de derecha a izquierda.
`lista.append(4)`	Añade el elemento `4` al final de la lista.
`lista[j] = z`	Redefine el valor del elemento en la posición `j` de la lista a `z`.
`lista.pop(k)`	Retorna el elemento en la posición `k` y lo elimina de `lista`. Sin parámetros retorna y elimina el último elemento.
`lista.count(c)`	Retorna la cantidad de apariciones del elemento `c`.
`lista.index(d)`	Retorna la posición del elemento `d`.
`lista.remove(e)`	Elimina la primera aparición del elemento `e`.
`lista.insert(i, f)`	Inserta el elemento `f` en la posición `i`[b].
`lista.sort()`	Ordena `lista` en orden creciente.
`lista1 + lista2`	Retorna una lista que concatena `lista1` y `lista2`.
`lista * n`	Retorna una lista que concatena `lista` `n` veces[c].

[a] Con `n` equivalente al largo de la lista.
[b] Los elementos siguientes son desplazados en 1 posición a la derecha.
[c] `n` debe ser entero.

Largo de una Secuencia

El largo de la secuencia `seq` se obtiene mediante la función nativa `len(seq)`. Se entiende por "largo" de una secuencia como el total de elementos que hay en esta.
En el caso de una lista, si tiene como elemento otra lista, este elemento sigue contando como uno solo.

Cortes y Copias

Pueden accederse cortes de una secuencia como una lista o string mediante la notación slice. Su notación básica es

`objeto[a:b:c]`,

retornando un objeto del mismo tipo del seccionado, con `a` el índice inicial del corte, `b` el índice final (él último índice retornado será siempre el mayor valor posible menor a `b`) y `c` la distancia entre dos elementos consecutivos recuperados. `a` y `b` deben ser índices válidos y `c` debe ser un número entero (positivo o negativo).
Esta notación tiene las siguientes variaciones:

- Si se omite `a` (pero no `:`), inicia desde el comienzo.
- Si se omite `b` (pero no `:`), llega hasta el final.
- Si se omite `c` (incluyendo o no `:`), se asume 1.

Consultas sobre Strings

Todos estos métodos devuelven `True` o `False`.

`s.isupper()`	Todas las letras son mayúsculas y hay al menos una.
`s.islower()`	Todas las letras son minúsculas y hay al menos una.
`s.isalpha()`	Todos los caracteres son alfabéticos.
`s.isdigit()`	Todos los caracteres son dígitos.
`s.isalnum()`	Todos los caracteres son alfabéticos o dígitos.
`s.endswith(t)`	El string termina con `t`.
`s.startswith(t)`	El string empieza con `t`.

String a Lista y Viceversa

- `s.split(sep)` separa el string utilizando `sep` como separador y retorna una lista cuyos elementos son los fragmentos del string.
- `sep.join(lista)` une los elementos de `lista`, separados por `sep`, en un único string. Solo funciona si los elementos son strings.

Strings

`s = "Monty"` `s = 'Python'`	Define un objeto de tipo string.
`s.find(sub)`	Retorna el índice donde empieza `sub`.
`s.rfind(sub)`	Como `find`, pero desde la derecha.
`s.index(sub)`	Como `find`, pero arroja error si no encuentra.
`s.lower()`	Retorna el string en minúscula.
`s.upper()`	Retorna el string en mayúscula.
`s.strip(t)`	Retorna el string eliminando los caracteres en `t` de los extremos del string.
`s.strip()`	Como `strip`, pero eliminando espacios en blanco.
`s.capitalize()`	Retorna el string convertido con el primer caracter a mayúscula, si es una letra, y el resto a minúscula.
`s.title()`	Retorna el string en "Formato De Título".
`s.count(sub)`	Retorna la cantidad de apariciones no superpuestas de `sub`.

Funciones Nativas

Estas funciones, entre otras, están siempre disponibles:

`abs(x)`	Valor absoluto de `x`.
`max(seq)`	Retorna el máximo de `seq`.
`min(seq)`	Retorna el mínimo de `seq`.
`range(stop)`	Genera una secuencia de
`range(i, f, s)`	valores enteros.
`round(x)`	Redondea `x` a entero o a
`round(x, n)`	`n` decimales.
`sorted(seq)`	Retorna la secuencia `seq` ordenada
`type(o)`	Retorna el tipo de `o`
`isinstance(o, c)`	Verifica si `o` es del tipo `c`

El nombre de cada tipo de dato es la función para cambiar a dicho tipo de dato, p.e.: `x = int(y)` o `l = list(s)` (si `s` es un conjunto o secuencia, como un string).

Funciones Importadas

Un módulo es un archivo que contiene definiciones y declaraciones en Python.
Existen módulos estándar (siempre disponibles para importar), definidos por el programador y externos (instalados en el sistema).
Para importar, utilizamos:

```
import module
# Uso de, por ejemplo, func1:
module.func1(params)
# Alternativamente, podemos dar un
# alias
import module as bla
bla.func1(params)
```

En casos especiales, podemos importar solo las funciones de interés utilizando:

```
from module import func1, func2, func3
```

De modo que se pueda utilizar `func1`, `func2` y `func3` como si fuesen nativas.

Definición de Funciones Propias

La estructura básica de definición de funciones es:

```
def func_name(arg1, arg2, arg3=valor):
    definiciones_y_operaciones_locales
    return resultado
```

- Dondequiera que se encuentre `return`, se finaliza la función y se entrega ese resultado. Puede haber varios, según si los resultados son condicionados.
- Los parámetros son tantos como se requieran. Si no se requieren, los paréntesis van vacíos.
- Los parámetros opcionales se escriben al final, dándoles su valor por defecto (como en `arg3`).

Una vez definida, se llama como una función nativa normal:

```
# Solo parámetros obligatorios
r = func_name(x, y)
# Con el parámetro opcional
r = func_name(x, y, z)
# Dando nombre del parámetro opcional
r = func_name(x, y, arg3=z)
```

Recursión

Python solo requiere el nombre de la función para poder utilizarla, por lo que podemos construir una función en base a sí misma:

```
def func(x):
    if condición(x):
        new_x = operaciones(x)
        return func(new_x)
    else:
        return otras_operaciones(x)
```

Comentarios y Documentación

Las líneas comenzadas por # son comentarios e ignorados por el intérprete. Sirven para documentar el código para quien lo lea.
Las funciones se pueden documentar con *docstrings*, explicando su utilidad y forma de uso inmediatamente después de definirlas:

```
def ejemplo(arg1, arg2=valor):
    """Descripción de lo que hace la
    función

    Entradas: Se describen sus entradas
    (incluyendo tipo de dato)
    Salidas: Se describen sus salidas
    (incluyendo tipo de dato)
    """
    operaciones
    return resultado
```

Las docstrings son un tipo de string multilínea, pueden escribirse con triple comilla simple (`'''así'''`) o triple comilla doble (`"""así"""`).
N.B.: Los comentarios son para quien leerá o modificará el código, las docstrings son para quien utilizará la función en el suyo propio.

Módulos Estándar

`math`	Funciones matemáticas.
`sys`	Funciones y parámetros del intérprete.
`op.path`	Manipulaciones de rutas de archivo.
`random`	Generación de números pseudo aleatorios.
`datetime`	Tipos básicos de fecha y hora.
`copy`	Operaciones de copia superficial y profunda.
`time`	Operaciones de tiempo.

Importación de Módulos Propios

Si escribimos funciones y constantes en un archivo independiente, puede importarse como cualquier otro módulo:

```
# Si utilidades.py existe en la
# misma carpeta
import utilidades
```

Alternativamente,

```
from utilidades import func
```

Se pueden importar nombres de funciones y constantes.
El nombre del módulo no puede contener espacios.

Ciclos con `for`

Repite la acción por cada elemento de la secuencia:

```
for elemento in secuencia:
    acciones_a_repetir
```

La variable `elemento` es definida en está secuencia y su valor es cada elemento de la secuencia en orden.
Entre las secuencias, se incluyen (entre muchos otros) archivos, `list`, `str`, `range`, etc.

Función `range`

Generación de secuencias numéricas:

- `range(stop)` genera números de 0 a `stop-1`.
- `range(start, stop, d)` genera números desde `start` hasta `stop-1`, con distancia de `d` entre ellos.

Nótese que `range` no es una lista, pero puede convertirse a una mediante `list(range(n))`.

Archivos

Un archivo es un objeto especial en Python creado con la función nativa `open`:

```
file = open(ruta, modo)
```

El parámetro `ruta` (`str`) especifica dónde encontrar el archivo (carpetas y nombre).
Cuando dejamos de usar el archivo, debemos cerrarlo, para lo que se utiliza el método `close`, como en `file.close()`.

Parámetros de archivo

Pueden darse tres modos de apertura:

Opción	Modo
`r`	Lectura
`w`	Escritura
`a`	Añadir

Si la codificación del archivo no es la del sistema, se puede especificar con `encoding`:

```
file = open(ruta, modo,
            encoding="utf8")
```

Las principales codificaciones aquí son `utf8` y `latin1`.

Operación `with`

Podemos utilizar un administrador de contexto para trabajar con archivos, a través del bloque `with ... as...`, en cuyo caso no es necesario utilizar `close`:

```
with open(ruta, modo) as file:
    operaciones_sobre(file)
```

Operaciones sobre Archivos

Los métodos básicos de archivo son:

`file = open(r, m)`	Abre el archivo en la ubicación `r` con modo `m`.
`file.read()`	Lee todo el archivo o
`file.read(n)`	hasta `n` caracteres.
`file.readline()`	Lee como string hasta el próximo final de línea.
`file.readlines()`	Lee como lista de strings todo el archivo.
`file.write(s)`	Escribe el string `s` en el archivo.
`file.writelines(l)`	Escribe la lista de strings `l` en el archivo.
`file.close()`	Cierra el archivo, impidiendo futuros accesos.

Cada lectura y escritura se hacen a partir de donde terminó la anterior. Toda lectura y escritura es literal: no se añaden ni quitan caracteres.

Módulos externos

Se pueden instalar módulos utilizando `pip` en la terminal del sistema:

```
pip install módulo
```

Algunos módulos importantes:

`numpy`	Arreglos multidimensionales y computación científica.
`matplotlib`	Visualizaciones y gráficos.

Una vez instalados, se importan como cualquier otro módulo:

```
import numpy as np
```

Las próximas secciones asumirán que `numpy` ha sido importado de esta manera.

Arreglos en `numpy`

Se puede transformar una secuencia (o secuencia de secuencias) mediante la función:

```
a = np.array(secuencia)
```

Todos los elementos del arreglo son del mismo tipo. El parámetro opcional `dtype` permite especificar el tipo de dato (e.g. , `dtype=float` para un arreglo de flotantes).
El tipo de dato del arreglo es `np.ndarray` y es mutable (soporta asignación de elementos).

Operaciones de arreglo

Siendo `u` un arreglo,

`u[i]`	Posición `i`.
`u[i, j]`	Posición en fila `i` y columna `j`
`u.shape`	Tupla con las dimensiones.
`len(u)`	Tamaño de primera dimensión de `u`.
`u.flat`	Iterador undimensional con todos los elementos.
`u.flatten()`	Copia unidimensional del arreglo.

Para acceder a los elementos, tanto `i` como `j` pueden ser cortes (*slices*)

Operaciones vectoriales

Todos los operadores aritméticos básicos y de comparación se aplican elemento a elemento en arreglos del mismo tamaño, p.e.:

```
w = u + v
```

Aplica la suma a cada par de elementos. Similarmente, si `a` es un escalar,

```
v = a + u
```

suma su valor a todos los elementos de `u`.
Para las operaciones lógicas vectoriales, existen los siguientes operadores:

`~u`	Negación por elementos.
`u & v`	Y lógico elemento a elemento.
`u | v`	O lógico elemento a elemento.
`u ^ v`	O excluyente elemento a elemento.

Estas operaciones tienen la misma precedencia que un comparador.

Funciones vectoriales

Las siguientes funciones son versiones para aplicar sobre todos los elementos de un array:

`np.sin(u)`	Seno.
`np.cos(u)`	Coseno.
`np.exp(u)`	Exponencial.
`np.log(u)`	Logaritmo.
`np.logical_not(u)`	`~u`
`np.logical_and(u, v)`	`u & v`
`np.logical_or(u, v)`	`u | v`
`np.logical_xor(u, v)`	`u ^ v`
`np.all(u)`	`True` si todos lo son.
`np.any(u)`	`True` si alguno lo es.

Generadores

Las siguientes funciones generan arreglos siguiendo diferentes patrones:

`np.arange(n)`	
`np.arange(a, b, c)`	Arreglo como `range`.
`np.linspace(a, b, c)`	`c` puntos desde `a` hasta `b`.
`np.zeros(n)`	Arreglo de 0 de tamaño `n`.
`np.ones(n)`	Arreglo de 1 de tamaño `n`.

Módulo Matplotlib

El módulo Matplotlib permite visualizar y crear gráficos a través de su interfaz `pyplot`:

```
import matplotlib.pyplot as plt
```

Las funciones principales se obtienen a partir de este submódulo. Tras generar los gráficos, estos se visualizan usando:

```
plt.show()
```

Gráfico de línea

Se crean gráficos de líneas con `plt.plot`. Su parámetro son arreglos o equivalentes con valores numéricos:

```
# Gráfico de y versus sus índices
g = plt.plot(y)
# Gráfico de y versus x
g = plt.plot(x, y)
# Gráfico de y versus x rojo
g = plt.plot(x, y, 'r')
```

Retorna una lista de gráficos.
Pueden graficarse varias líneas agrupando dos o tres parámetros: eje x, eje y, formato (opcional):

```
g = plt.plot(x_1, y, x_1, z, x_2, f)
```

Estilo de las líneas

El estilo del gráfico se controla mediante la función `plt.setp` y sus parámetros:

```
plt.setp(g, param_1=valor,
         param_2=valor,...)
```

El parámetro `g` es un gráfico o lista de gráficos. El resto de parámetros incluye:

`color`	Color de la línea.
`marker`	Tipo de marcador.
`markersize`	Tamaño de marcador en puntos.
`linestyle`	Tipo de línea.
`linewidth`	Ancho de línea en puntos.

Gráfico Circular

Se genera con `plt.pie` y requiere un vector o lista con valores. Genera una "torta" de radio 1. Sus parámetros incluyen:

`autopct`	`str` para etiquetar las porciones con su valor.
`labels`	Secuencia de etiquetas.
`explode`	Secuencia con distancia del centro para cada porción.
`labeldistance`	Distancia de la etiqueta.
`rotatelabels`	Para `True`, rota las etiquetas para cada porción.

Ejemplo:

```
valores = [0, 2, 1, 3]
etiquetas = ["a", "b", "c", "d"]
plt.pie(valores, labels=etiquetas,
        autopct="%1.f%%")
```

Histograma

Muestra frecuencia de ocurrencia de valores en contenedores (*bins*). Su parámetro `bins` es cuántos contenedores hay o una secuencia que los especifique:

```
dados = np.random.randint(1, 10, 10000)
plt.hist(dados, bins=5)
```

Gráfico de Barras

Un gráfico de barras se construye igual que el de líneas, pero solo uno a la vez. Sus parámetros principales son posición y alto de cada barra:

```
plt.bar(np.arange(len(valores)),
        valores, width=0.8,
        align='center')
```

Los valores por defecto de los parámetros son los dados.
Para un gráfico de barras horizontal, se usa `plt.barh` y se intercambian los parámetros height y width:

```
plt.barh(np.arange(len(valores)),
         valores, height=0.8,
         align='center')
```

Etiquetas del gráfico

```
plt.title("Título del gráfico")
plt.xlabel("Eje $x$")
plt.ylabel("Eje $y$")
# Posiciones y etiquetas deben
# ser del mismo largo
# Etiquetas pueden ser strings
plt.xticks(posiciones_x, etiquetas_x)
plt.yticks(posiciones_y, etiquetas_y)
```

Leyenda

Las leyendas se incluyen a partir de las etiquetas de cada gráfico (parámetro opcional `label`) o directamente con `plt.legend`.

```
# Genera leyenda a partir de etiquetas
# previas
plt.legend()
# Explícitamente se dan las leyendas
# (en orden de creación)
plt.legend(["a", "b", "c"])
# Asignando explícitamente a cada uno
plt.legend(g, "a")
```

Múltiples gráficos

```
# Crea y selecciona figura en blanco
fig = plt.figure()
# Selecciona figura n
fig = plt.figure(n)
# Selecciona el subgráfico i
# al subdividir la figura en n por m
plt.subplot(n, m, i)
```

Se crean los gráficos en la última figura seleccionada.

```
# Crea figura y un eje para graficar
fig, ax = plt.subplots()
ax.plot(x, y)
# Crea figura y arreglo de n ejes
# para graficar
fig, ax = plt.subplots(n)
ax[0].plot(x, y)
# Crea figura y matriz de
# n por m de ejes
fig, ax = plt.subplots(n, m)
ax[0, 0].plot(x, y)
```