Creación y Uso de Funciones

Computación en la Biología

17/11/2025

Docente: Lic. Joaquín Pereira

Introducción | *Aclaraciones Iniciales*

Mauricio Langleib

Joaquín Pereira

Juan Trinidad

Héctor Romero

Clase 01: Estructuras de datos en Python. Datos numéricos, cadenas de caracteres, listas, conjuntos y diccionarios.

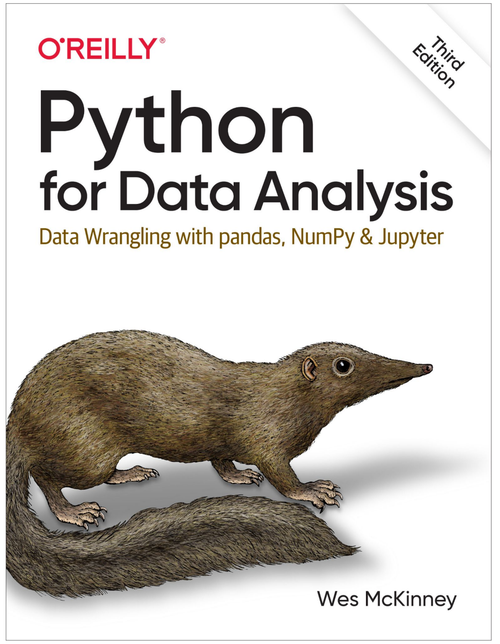
Clase 02: Control de flujo. Operaciones lógicas, uso de condicionales y bucles. Introducción a *list comprehensions*.

Clase 03: Definición de funciones y el alcance de variables. Presentación de los módulos '*sys*', '*os*', '*re*', '*glob*' y '*subprocess*'. Creación de scripts y el uso de '*argparse*' para manejar argumentos desde la línea de comandos.

Clase 04: Librería Pandas y los DataFrames. Importación y exploración de datos desde archivos CSV/TSV.

Clase 05: Introducción a la visualización de datos con 'seaborn'.

Ningún docente es ingeniero en computación… tenemos una aproximación pragmática al tema !



Introducción | Repase de flujos de trabajo y bucles

```
x = -5
if x < 0:
    print("It's negative")
```

```
if x < 0:
    print("It's negative")
elif x == 0:
    print("Equal to zero")
elif 0 < x < 5:
    print("Positive but smaller than 5")
else:
    print("Positive and larger than or equal to 5")
```

```
In [130]: a = 5; b = 7

In [131]: c = 8; d = 4

In [132]: if a < b or c > d:
   .....:     print("Made it")
Made it
```

Introducción | Repase de flujos de trabajo y bucles

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

Funciones | *Porqué usamos funciones?*

Las funciones nos permiten **reusar** nuestro código de manera sencilla y clara

Estas ayudan a que nuestro código sea más **modular**

Lo cual facilita su **lectura y debugeo**

Funciones | *Funciones definidas por el Usuario*

Como se definen las funciones?

```
def function_name(parameters):
    """Optional docstring describing what the function does."""
    # Function body - indented code block
    statement1
    statement2
    # ...
    return result  # Optional return statement
```

```
var1 = function_name(new_parameters)
```

Se definen con keyword ***def***, *seguidos del* ***nombre*** *de la función,* mejor si está en minúsculas y separado por guiones (**PEP8**). Luego entre paréntesis van los parámetros de las función y dentro del bloque de indentación un **docstring** para dar contexto seguido del cuerpo (**código**) de la función. Finalmente se usa el keyword **return,** para devolver un resultado

Pero también es posible definir un función sin parámetros, docstring, return . . .

```
def minimal_function():
    # Some code
```

Funciones | *Argumentos posicionales y keywords*

Las funciones pueden tener **argumentos posicionales** o definidos mediante palabras clave **(keywords)**

Los argumentos posicionales:

1) Escribirse en orden o llamarse de manera literal
2) Son obligatorios

Los argumentos por keywords:

1) No son obligatorios de usar
2) Van siempre después que los args. posicionales

```
def compute_xy(x, y, divide=False):
    if divide:
        return x/y
    else:
        return x - y
```

```
compute_xy(6,2)
```

✓ 0.0s

```
4
```

```
compute_xy(6,2, divide=True)
```

✓ 0.0s

```
3.0
```

Funciones | *Argumentos de largo variable*

También se pueden definir argumentos de largo variable como los son: *args y **kargs

Los *args son usados para darle más flexibilidad a las funciones

Colectan argumentos en una tupla

```
def total(*numbers):
    return sum(numbers)

total(1, 2, 3, 4)
```

✓ 0.0s

```
10
```

```
print('a', 'b', 'c')
```

✓ 0.0s

```
a b c
```

Los *kargs se pueden usar para configurar funciones internas dentro de una misma función.

Colectan keywords en un diccionario

```
def show_info(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

show_info(name="Ana", age=28)
```

✓ 0.0s

```
name Ana
age 28
```

Funciones | *Namespace* y *Alcance de variables*

El ***namespace*** se refiere al mapa que existe entre los **objetos** de python y sus respectivos **nombres**

Puedes pensarlo como un diccionario entre los objetos y su nombre

Dentro del namespace se encuentran:
- Funciones nativas de python: print(), sum(), abs(), . . .
- Funciones definidas por el usuario
- Variables definidas por usuario: var1= “Hola”, my_list=[1,2,3], . . .
- Variables globales del sistema

Funciones | *Namespace* y *Alcance de variables*

Las variables **definidas dentro** de una función, solo existen en el namespace mientras la función se ejecuta

```
def func():
    a=1
    return a+1

func()
```

✓ 0.0s

```
2
```

```
type(a)
```

✓ 0.0s

```
---------------------------------------
NameError
Cell In[14], line 1
----> 1 type(a)

NameError: name 'a' is not defined
```

Sin embargo, las variables previamente definidas, pueden ser accedidas, (y en ocasiones alteradas) por el código ejecutado dentro de una función.

```
a = 'Hello!'

def func():
    return a * 2

func()
```

✓ 0.0s

```
'Hello!Hello!'
```

```
a = []

def func():
    a.append('Adios!')
    return a

func()
print(a)
```

✓ 0.0s

```
['Adios!']
```

Funciones | *Namespace* y *Alcance de variables*

Las variables definidas dentro de una función (**local variables**), solo existen en el namespace mientras la función se ejecuta

```
def func():
    a=1
    return a+1

func()
```

✓ 0.0s

```
2
```

```
type(a)
```

✓ 0.0s

```
---------------------------------------------------------------------------
NameError
Cell In[14], line 1
----> 1 type(a)

NameError: name 'a' is not defined
```

Sin embargo, las variables previamente definidas, pueden ser accedidas, (y en ocasiones alteradas) por el código ejecutado dentro de una función.

```
a = 'Hello!'

def func():
    return a * 2

func()
```

✓ 0.0s

```
'Hello!Hello!'
```

```
a = []

def func():
    a.append('Adios!')
    return a

func()
print(a)
```

✓ 0.0s

```
['Adios!']
```

Aunque normalmente si una variable **local** y **global** comparten el mismo nombre estas no se sobreescriben

```
x = 10        # global

def show():
    x = 5     # local
    print("Inside:", x)

show()
print("Outside:", x)
```

✓ 0.0s

```
Inside: 5
Outside: 10
```

En resumen, las mejores prácticas incluyen:

- Mejor devolver una variable nueva que modificar una global
- Evitar sobreescribir objetos existentes (list, dict, etc.)
- Usar docstring y nombres descriptivos
- Para scripts grandes agrupa el código en funciones y módulos

Las funciones **Lambda**, también llamadas funciones anónimas, se usan para crear **funciones cortas**, sin necesidad de polucionar el namespace.

Comúnmente usadas como argumentos para otras funciones

```python
def aplicar_f(x,f):
    y = f(x)
    return y

aplicar_f(10, lambda x: x + 1)
```

✓ 0.0s

```
11
```

Las funciones **Lambda**, también llamadas funciones anónimas, se usan para crear **funciones cortas**, sin necesidad de polucionar el namespace.

Comúnmente usadas como argumentos para otras funciones

```python
def aplicar_f(x,f):
    y = f(x)
    return y

aplicar_f(10, lambda x: x + 1)
```

✓ 0.0s

```
11
```

Las funciones **Lambda**, también llamadas funciones anónimas, se usan para crear **funciones cortas**, sin necesidad de polucionar el namespace.

Comúnmente usadas como argumentos para otras funciones

```python
def aplicar_f(x,f):
    y = f(x)
    return y

aplicar_f(10, lambda x: x + 1)
```

✓ 0.0s

```
11
```

Funciones | *Las funciones se pueden importar... algo muy útil!*

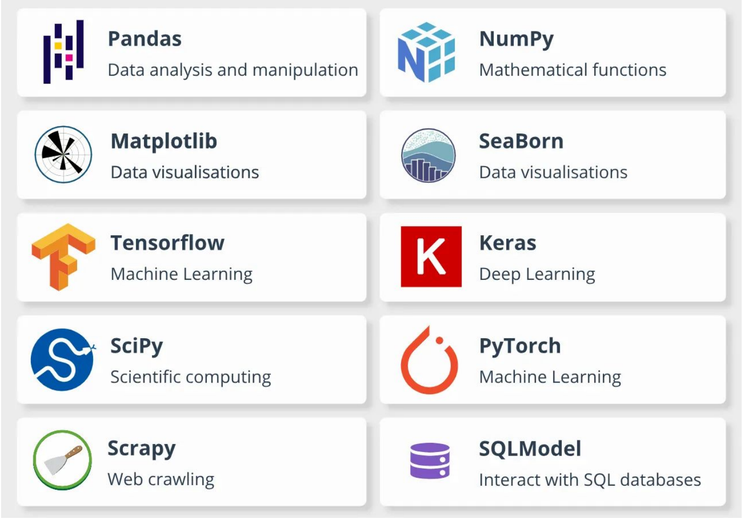
```python
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

```python
import some_module
result = some_module.f(5)
pi = some_module.PI
```

```python
from some_module import g, PI
result = g(5, PI)
```



Podemos guardar funciones y variables en un archivo

Esto es bastante útil, porque luego las podemos **importar**

Esta es la base para que se puedan formar librerías con funciones, clases, módulos y demás ya definidas.

Así, nos ahorramos el reinventar la rueda

Vamos a abrir el Jupyter notebook **creacion_funciones.ipynb**

Módulos para el Scripting

Computación en la Biología

17/11/2025

Docente: Lic. Joaquín Pereira

Los **módulos nativos** de python tienen un gran potencial, permitiendo extender en gran medida las capacidades del lenguaje

A partir de estos es posible:

- Interactuar con el sistema operativo (os, sys)
- Manejo de rutas y archivos (os, path)
- Procesamiento de texto (glob, re)
- Argumentos e interacción desde línea de comandos (argparse, subprocess)

Usos comunes:

- Listar archivos
- Crear directorios
- Leer argumentos desde CLI
- Buscar patrones en texto o en archivos FASTA
- Ejecutar comandos como `grep`, `ls`, `blastn`, etc.

Principales ventajas:

- No requieren instalación
- Funcionan igual en Linux, macOS y Windows
- Son seguros y bien mantenidos

Funciones | *Módulo sys*

El módulo **sys** permite interactuar directamente con el entorno de ejecución

Se usa comúnmente para pasar **argumentos** a scripts ejecutados desde la terminal, empleando el método ***argv***

También se usan para terminar la ejecución del script (***exit***) y redirigir los flujos de salida (***stderr***, ***stdout***)

```python
# demo_sys.py
import sys

print("Argumentos recibidos:", sys.argv)

if len(sys.argv) < 2:
    print("Debes pasar un nombre", file=sys.stderr)
    sys.exit(1)

print(f"Nombre del Script: {sys.argv[0]}")
print(f"Hola {sys.argv[1]}!")
print(f"Hello {sys.argv[2]}!")
```

```
python demo_sys.py Joaquin Smith
```

```
Argumentos recibidos: ['demo_sys.py', 'Joaquin', 'Smith']
Nombre del Script: demo_sys.py
Hola Joaquin!
Hello Smith!
```

Funciones | *Módulo os*

El módulo **os** es muy potente, ya que permite interactuar con las variables, carpetas y archivos del sistema operativo

- Obtener directorio actual, os.getcwd()
- Listar archivos, os.listdir()
- Crear y borrar carpetas, os.mkdir(), os.remove()
- Manejar rutas, `os.path`.
- Comprobar existencia de rutas, os.path.exist()

```python
import os

print("Directorio actual:", os.getcwd())
print("Contenido:", os.listdir("."))

ruta = os.path.join("data", "archivo.txt")
print("Ruta creada:", ruta)

# Crear carpeta si no existe
os.makedirs("resultados", exist_ok=True)
```

```python
import os

carpeta = "."
for nombre in os.listdir(carpeta):
    if nombre.endswith(".txt"):
        print("Encontrado:", nombre)
```

Funciones | *Clase Path*

La clase **Path** de la librería pathlib, es construido a partir del módulo os, la cual se enfoca en el manejo de rutas

Teniendo una sintaxis más limpia y pythonica

```python
from pathlib import Path
p = Path('new_dir')

# Creación clara de nuevos paths
f = p / "file1.txt"
```

```python
from pathlib import Path
p = Path("data/archivo.txt")
```

```python
print(p.parent)  # data
print(p.stem)    # archivo
print(p.suffix)  # txt
```

```python
p.exists() # Verifica existencia
p.is_dir() # Verifica si es directorio
p.is_file() # Verifica si es archivo
```

```python
d=Path("dir1/dir2")

# Crea todos los directorios de la ruta
# No genera errores si ya fueron creados
p.mkdir( exist_ok=True, parents=True)
```

Funciones | Módulo glob

El módulo **glob** permite encontrar archivos mediante el uso de patrones

* : Cualquier patrón

?: Cualquier carácter

**: Cualquier patrón de manera recursiva

La clase **Path**, también tiene su propio globing

```python
for archivo in Path("data").iterdir():
    print(archivo)

for tsv in Path(".").glob("*.tsv"):
    print(tsv)
```

```python
import glob

for archivo in glob.glob("*.py"):
    print(archivo)
```

```
main.py
utils.py
analisis.py
```

```python
for f in glob.glob("data/*.fasta"):
    print("FASTA:", f)
```

```python
for f in glob.glob("data/**/*.txt", recursive=True):
    print("TXT:", f)
```

Funciones | Módulo re

El módulo `re` permite trabajar con **expresiones regulares** en Python. Con estas se pueden **buscar, validar, extraer o reemplazar** patrones de texto

`.` → cualquier carácter

`\d` → dígito

`\w` → letra, número o guión bajo

`\s` → espacio

`+` → una o más veces

`*` → cero o más veces

`?` → opcional

`[abc]` → cualquiera de esos caracteres

`^` → inicio del texto

`$` → final del texto

```python
import re
# Busca la primer coincidencia
re.search(r"\d+", "Edad: 32") # encuentra "32"
```

```python
# Coincidencias solo al inicio del texto
re.match(r"\d+", "32 años")  # sí coincide
re.match(r"\d+", "Edad 32")  # no coincide
```

```python
# Busca todas las coincidencias, devuelve una lista
re.findall(r"\w+@", "pedro@gmail.com juan@uni.edu")
# ["pedro@", "juan@"]
```

```python
# Divide un string usando regexs
re.split(r",\s*", "a, b, c")
# ["a", "b", "c"]
```

```python
re.sub(r"\d+", "X", "Año 2025")
# "Año X"
```

Integración con la Terminal y Control de Errores

Computación en la Biología

17/11/2025

Docente: Lic. Joaquín Pereira

Funciones | Gestor de contexto with

`with` es un context manager, es decir, una estructura que se usa para manejar recursos de forma segura y automática, incluso si ocurre un error.

Ejemplos típicos incluyen

- abrir y cerrar archivos
- conexiones a bases de datos

```python
f = open("datos.txt")
contenido = f.read()
f.close()  # si ocurre un error, nunca se ejecuta
```

```python
with open("datos.txt") as f:
    contenido = f.read()
```

```python
def contar_fasta(path):
    count = 0
    with open(path) as f:
        for linea in f:
            if linea.startswith(">"):
                count += 1
    return count

print(contar_fasta("secuencias.fasta"))
```

Funciones | Módulo Subprocess

`subprocess` es un módulo que permite **ejecutar comandos del sistema operativo** desde un script de Python

- comandos de Linux (`ls`, `grep`, `wc`, `mkdir`, etc.)
- herramientas bioinformáticas (`blastn`,, `samtools`, `iqtree`, etc.)
- scripts bash

```python
import subprocess

subprocess.run(["echo", "Hola desde Bash"])
```

```python
res = subprocess.run(
    ["ls", "-l"],

    # Captura stdout y stderr en res.stdout y res.stderr
    capture_output=True,

    # Devuelve stdout y stderr como strings (no bytes)
    text=True
)

# salida estándar del comando
print(res.stdout)

# código de salida (0 = OK, distinto de 0 = error)
print(res.returncode)
```

Funciones | Módulo Subprocess

```python
res = subprocess.run(
    ["ls", "-l"],

    # Captura stdout y stderr en res.stdout y res.stderr
    capture_output=True,

    # Devuelve una excepción (error), si el comando no se ejcuta
    # de manera correcta
    check=True,

    # Devuelve stdout y stderr como strings (no bytes)
    text=True
)

# salida estándar del comando
print(res.stdout)

# código de salida (0 = OK, distinto de 0 = error)
print(res.returncode)
```

```python
subprocess.run([
    "blastn",
    "-query", "seq.fasta",
    "-db", "nt",
    "-out", "resultado.txt"
])
```

Funciones | Bloque Try, Except

Las **excepciones** son errores que ocurren durante la ejecución de un programa.

Cuando Python encuentra una situación inesperada (archivo inexistente, división por cero, tipo incorrecto, etc.), detiene el programa y lanza una ***exception***.

Las excepciones más comunes incluyen:

- ValueError,
- TypeError,
- FileNotFoundError,
- IndexError,
- KeyError,
- ZeroDivisionError,

```
open('algun/archivo')
```

0.0s

```
---------------------------
FileNotFoundError
Cell In[7], line 1
----> 1 open('algun/archivo')
```

Funciones | Bloque Try, Except

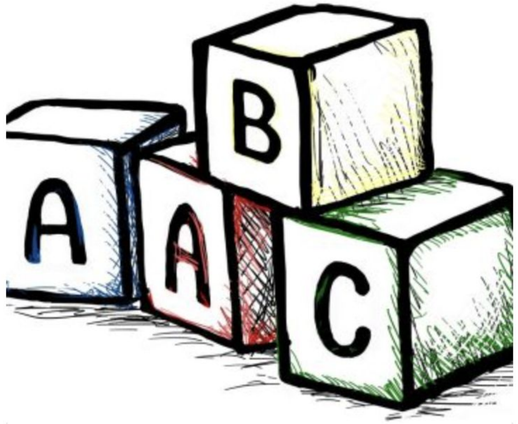
`try/except` es una estructura de Python que permite **manejar errores (excepciones)** sin que el programa se detenga de forma abrupta.

Sirve para que tu código sea **más robusto**, especialmente en scripts que leen archivos, procesan datos o ejecutan comandos externos.

- El bloque **`try`** contiene el código que *podría fallar*.
- El bloque **`except`** indica qué hacer *si ocurre un error.*

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
except Exception:
    print("Ocurrió un error desconocido.")
```

Algunos cursos basados en Python en UY



Aprendizaje Automático Básico
para Científicos (AABC) (Edición