Corso Pratico Python

[Modulo 1- 2026]

1. Introduzione all’ambiente Python
1.1. La sintassi : motivazioni al disegno del linguaggio

Richiedente: Dott. Roberto Felici [CNR-ISM] • Insegnante: Dott. Scigè J. Liu [INAF-IAPS]

1.1. La sintassi … le stringhe

Le stringhe [str()] sono aggregati di caratteri – parenti dei byte-array

```
str()con " ":
  str1 = "L'intelligenza artificiale non sostituirà l'estro umano."
str()con ' ':
  str2 = 'L\'intelligenza artificiale non sostituirà l\'estro umano.'
  str3 = 'ma con gli apici posso scrivere "citazioni"'
str()con """ … """:
  str4 = """
Con le triplici potrei:
 - Andare a capo
"""
```

1.1. La sintassi … le stringhe

Le stringhe [str()] sono aggregati di caratteri – parenti dei byte-array

```
str()con " ":
  str1 = "L'intelligenza artificiale non sostituirà l'estro umano."
str()con ' ':
  str2 = 'L\'intelligenza artificiale non sostituirà l\'estro umano.'
  str3 = 'ma con gli apici posso scrivere "citazioni"'
str()con """ … """:
  str4 = """
Con le triplici potrei:
 - Andare a capo
"""
```

```
str() con r' ':
  str5 = r’c:\cartella\windows.dir'
str() con f' ':
  str6 = f’apotema : {apotema:3f}'
```

1.1. La sintassi … le stringhe

Le stringhe [str()] sono aggregati di caratteri – parenti dei byte-array

```
str()con " ":
  str1 = "L'intelligenza artificiale non sostituirà l'estro umano."
str()con ' ':
  str2 = 'L\'intelligenza artificiale non sostituirà l\'estro umano.'
  str3 = 'ma con gli apici posso scrivere "citazioni"'
str()con """ … """:
  str4 = """
Con le triplici potrei:
 -
""
```

```
str() con r' ':
  str5 = r'c:\cartella\windows.dir'
```

```
apotema = 12.3456789
# Formattazione: 15 caratter, 6 decimali, tipo float (f)
percorso_file = fr"C:\Risultati\Dati_{apotema:15.6f}.csv"
```

1.1. La sintassi … i commenti

I commenti sono parti dei corpus non fungibili

```
Commenti con # (detto in-linea):
  str1 = "testo di prova" # il commento non partecipa all'assegnazione
Commenti con ## (detto documentato, alle volte con ! o <):
  mioPiGreco = np.arctan(1.0) * 4 ##< Calcolo di Pi Greco dinamicamente
  ##! Calcolo di Pi Greco dinamicamente
  mioPiGreco = np.arctan(1.0) * 4
```

– o interpretati dai plug-in

1.1. La sintassi … i commenti

I commenti sono parti dei corpus non fungibili

```
Commenti con # (detto in-linea):
  str1 = "testo di prova" # il commento non partecipa all'assegnazione
Commenti con ## (detto documentato, alle volte con ! o <):
  mioPiGreco = np.arctan(1.0) * 4 ##< Calcolo di Pi Greco dinamicamente
  ##! Calcolo di Pi Greco dinamicamente
  mioPiGreco = np.arctan(1.0) * 4
```

– o interpretati dai plug-in

```
Separatore di sezione #%%:
  usati nelle versioni «stream» per segmentare le parti del corpus,
  utili per esecuzione report-in-linea, a-blocchi o debug
#%% sezione degli import
import numpy as np
#%% sezione delle definizione di funzioni
def areaPoligono(lato, numLati, rapportoApotema):
    return lato*numLato*(lato*rapportoApotema)/2
```

1.1. La sintassi … i commenti

I commenti sono parti dei corpus non fungibili

Commenti con **#** (detto in-linea):

```
  str1 = "testo di prova" # il commento non partecipa all'assegnazione
```

Commenti con **##** (detto documentato, alle volte con ! o <):

```
  mioPiGreco = np.arctan(1.0) * 4 ##< Calcolo di Pi Greco dinamicamente

  ##! Calcolo di Pi Greco dinamicamente
  mioPiGreco = np.arctan(1.0) * 4
```

– o interpretati dai plug-in

Separatore di sezione **#%%:**

usati nelle versioni «stream» per segmentare le parti del corpus,

utili per esecuzione report-in-linea, a-blocchi o debug

```
#%% sezione degli import
import numpy as np
#%% sezione delle definizione di funzioni
                                    Apotema):
                                    otema)/2
```

Dichiarazione Interprete (#!)

```
  #!/usr/bin/env python3
```

Fattivamente interpretato dalla shell e non da python

1.1. La sintassi … i commenti

I commenti sono parti dei corpus non fungibili

Commenti con **#** (detto in-linea):

```
  str1 = "testo di prova" # il commento non partecipa all'assegnazione
```

Commenti con **##** (detto documentato, alle volte con ! o <):

```
  mioPiGreco = np.arctan(1.0) * 4 ##< Calcolo di Pi Greco dinamicamente

  ##! Calcolo di Pi Greco dinamicamente
  mioPiGreco = np.arctan(1.0) * 4
```

– o interpretati dai plug-in

Separatore di sezione **#%%:**

usati nelle versioni «stream» per segmentare le parti del corpus,

utili per esecuzione report-in-linea, a-blocchi o debug

```
#%% sezione degli import
import numpy as np
#%% sezione delle definizione di funzioni
                                    Apotema):
                                    otema)/2
```

Dichiarazione Interprete (#!)

```
  #!/usr/bin/env python3
```

Fattivamente interpretato dalla shell e non da python

1.1. La sintassi … i commenti

Riepilogo Tecnico

- **#!**: Inserito all'inizio per rendere lo script eseguibile su sistemi Unix.
- **#**: Usato per brevi spiegazioni del calcolo.
- **##**: Usato per evidenziare commenti [per posizione/enfasi].
- **#%%**: Usato per dividere (comodo se usi VS Code o PyCharm).

Corso Pratico Python

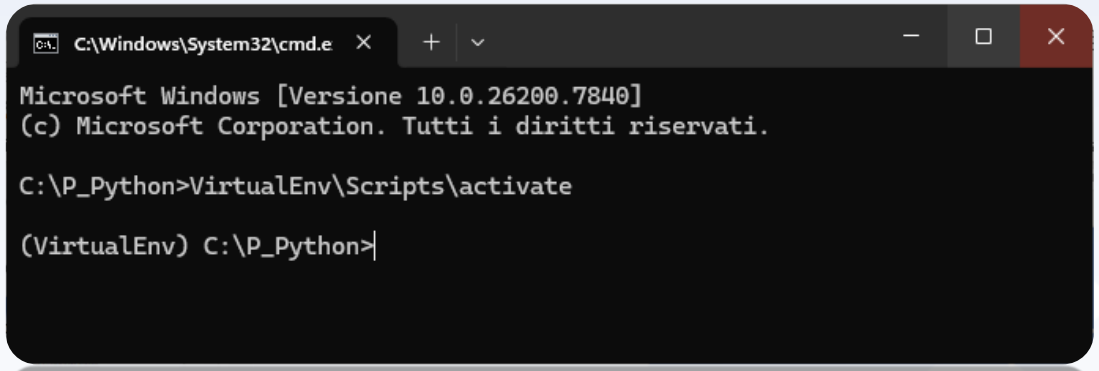
[Modulo 1- 2026]

1. Introduzione all'ambiente Python
1.2. Corpus, Moduli e Cartelle

Richiedente: Dott. Roberto Felici [CNR-ISM] • Insegnante: Dott. Scigè J. Liu [INAF-IAPS]

1.2. Corpus, Moduli e Cartelle

Corpus: elemento aggregato di elementi sintattici

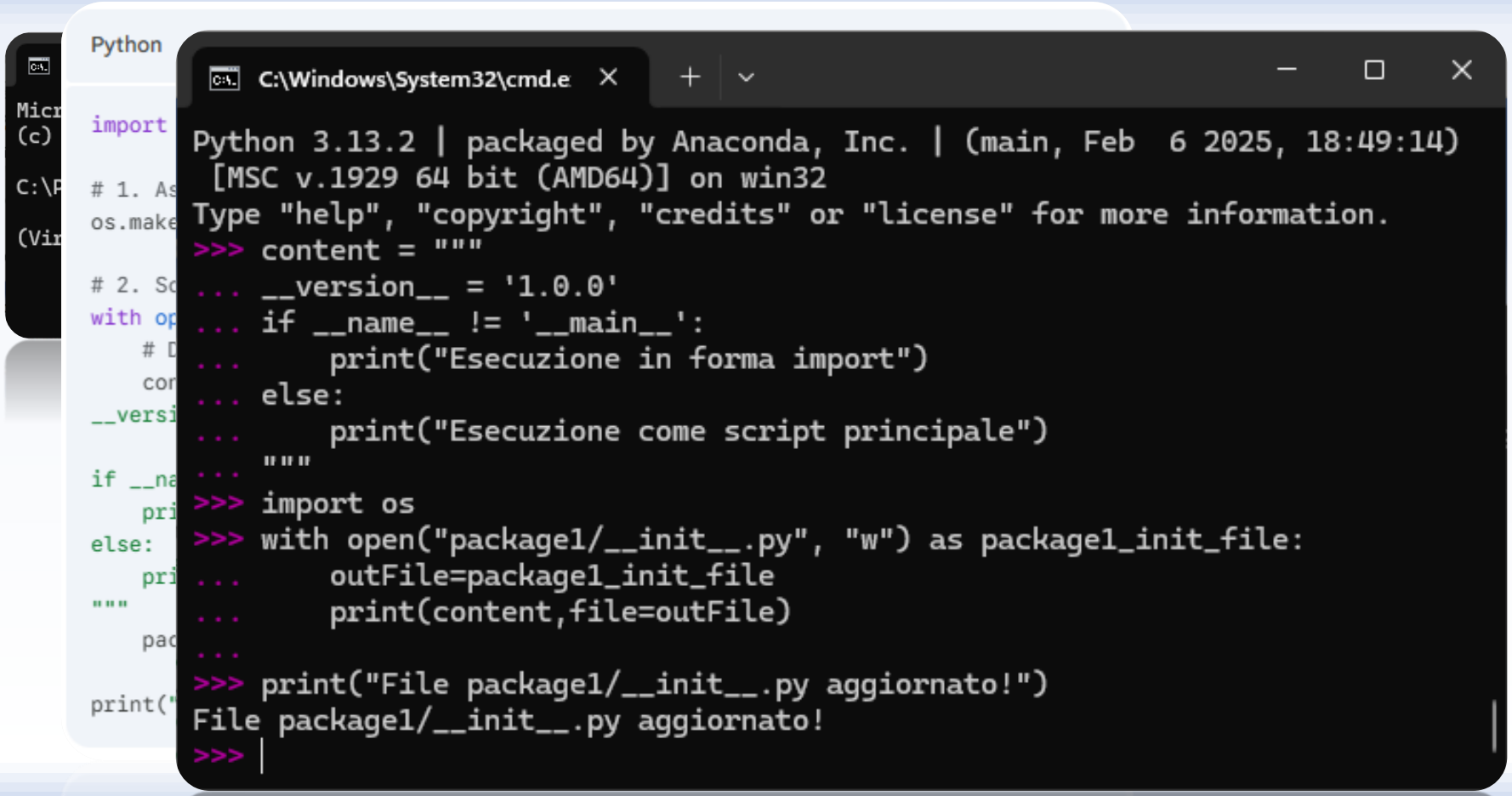


Intervallo Tecnico 1.2.1 – Command-Line 1

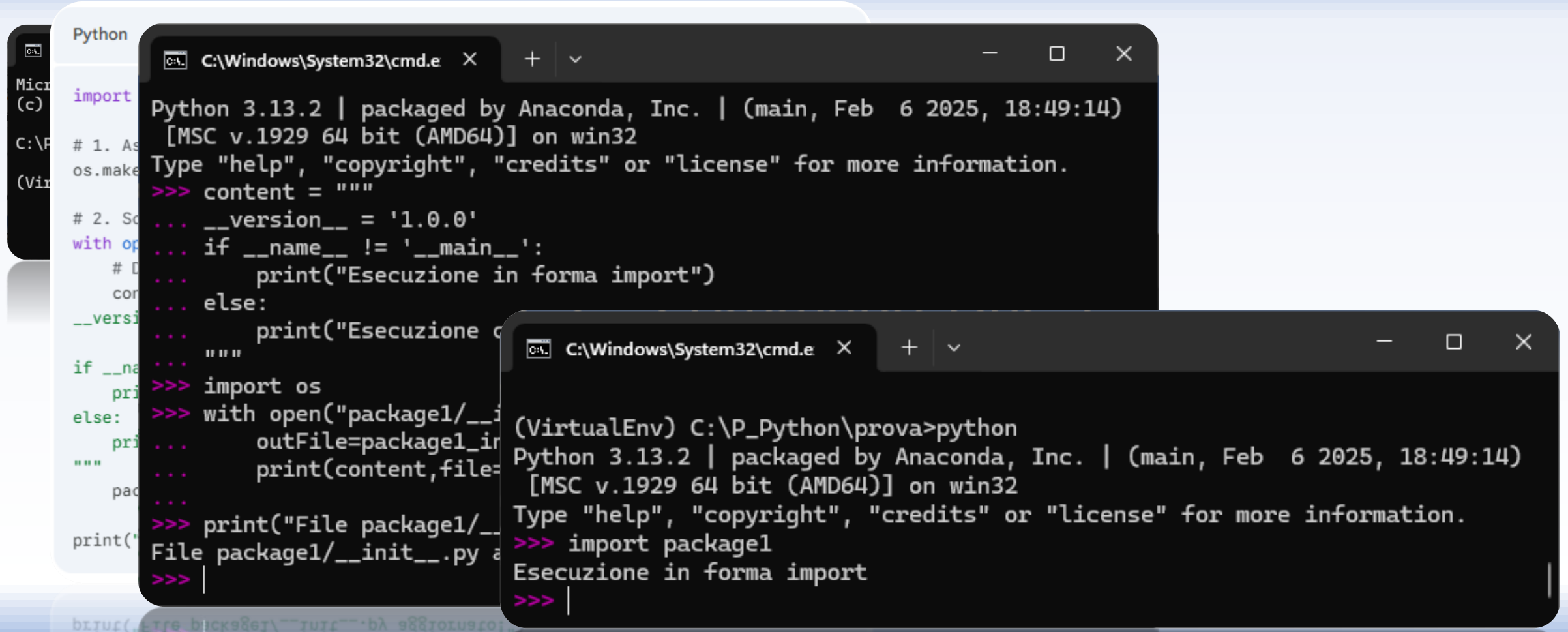
Python

```python
import os

# 1. Assicuriamoci che la cartella esista
os.makedirs("package1", exist_ok=True)

# 2. Scriviamo il contenuto nel file __init__.py
with open("package1/__init__.py", "w") as package1_init_file:
    # Definiamo il contenuto del file come stringa
    content = """
__version__ = '1.0.0'

if __name__ != '__main__':
    print("Esecuzione in forma import")
else:
    print("Esecuzione come script principale")
"""
    package1_init_file.write(content.strip())

print("File package1/__init__.py aggiornato!")
```

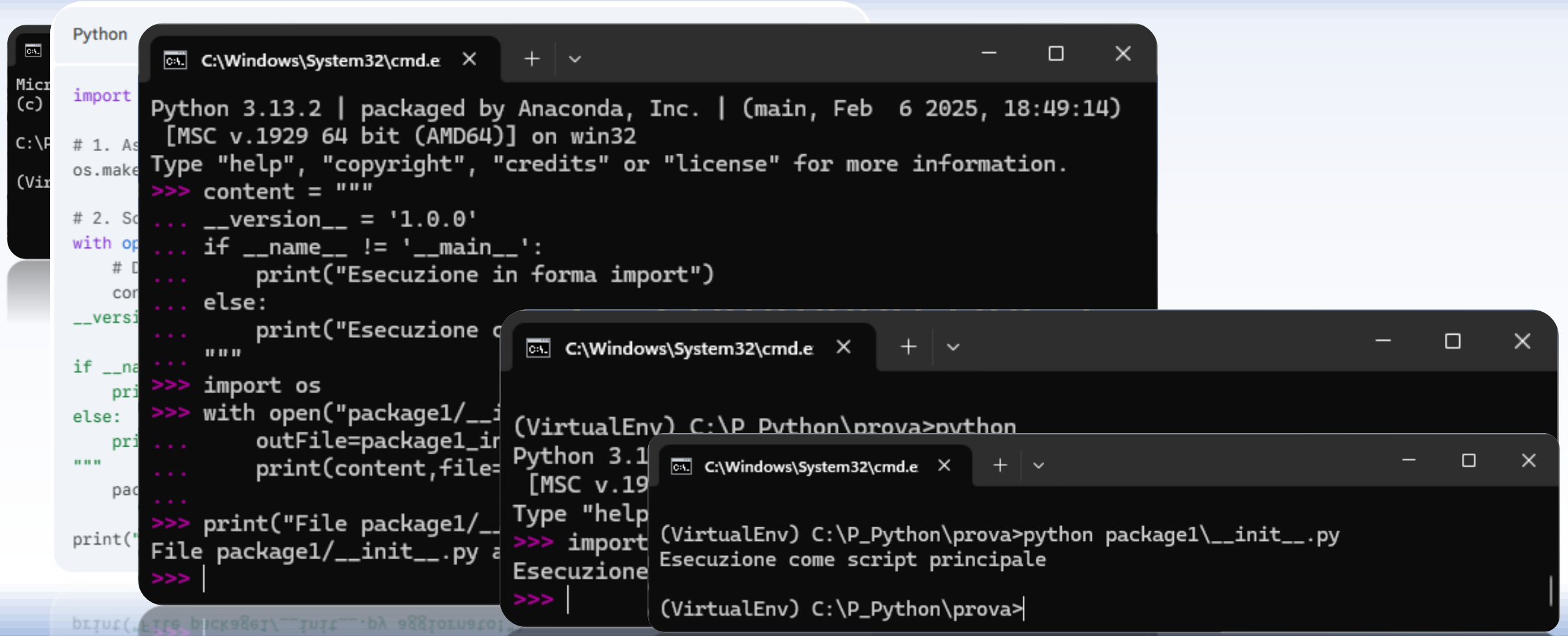
Intervallo Tecnico 1.2.1 – Command-Line 1



Intervallo Tecnico 1.2.1 – Command-Line 1



Intervallo Tecnico 1.2.1 – Command-Line 1



Intervallo Tecnico 1.2.2 – Command-Line 2

C:\P_Python\prova\package1__init__.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

__init__.py

```
1  #
2  __version__ = '1.2.3'
3  if __name__ != '__main__':
4      print("Esecuzione in forma import [__init__]")
5  else:
6      print("Esecuzione come script principale [__init__]")
7  #
```

__main__.py

```
1  #if inside package1:
2  from .  import __version__ as package1_version
3
4  __version__ = '4.5.6'
5  if __name__ == '__main__':
6      print("Esecuzione come script principale [__main__]")
7      print(f"package1_main    : {__version__}")
8      print(f"package1_version : {package1_version}")
9  #
```

Python file INS

C:\Windows\System32\cmd.e

```
(VirtualEnv) C:\P_Python\prova>python -m package1
Esecuzione in forma import [__init__]
Esecuzione come script principale [__main__]
package1_main    : 4.5.6
package1_version : 1.2.3

(VirtualEnv) C:\P_Python\prova>
```

1.2. Corpus, Moduli e Cartelle - esempio import 1

Cartella/File: ./geometria/formule.py e ./main.py

```
# cartella /geometria/
    # formule.py
    def area_rettangolo(base, altezza):
        return base * altezza
```

```
# cartella .
# main.py
from geometria.formule import area_rettangolo as ar
base = 10
altezza = 5
print(f"L'area calcolata è: {ar(base, altezza)}")



#
```

1.2. Corpus, Moduli e Cartelle - esempio import 2

Cartella/File: ./geometria/formule.py e ./main.py

```
# cartella /geometria/
    # formule.py
    def area_rettangolo(base, altezza):
        return base * altezza
```

```
# cartella .
# main.py
from geometria.formule import area_rettangolo as ar
base = '10'
altezza = 5
print(f"L'area calcolata è: {ar(base, altezza)}")



#
```

1.2. Corpus, Moduli e Cartelle - esempio import 3 + tipi

Cartella/File: ./geometria/formule.py e ./main.py

cartella /geometria/

```
# formule.py
def area_rettangolo(base: float, altezza: float) ->
float:
    """
    Calcola l'area di un rettangolo.
    :param base: La base del rettangolo (float)
    :param altezza: L'altezza del rettangolo (float)
    :return: L'area calcolata (float)
    """
    return base * altezza
```

cartella .

```
# main.py
from geometria.formule import
          area_rettangolo as ar
base = '10'
altezza = 5
print(f"L'area calcolata è:
                {ar(base, altezza)}")
#
```

1.2. Corpus, Moduli e Cartelle - esempio import 3 + casting

- **Arricchimento Contenuti: Gestione degli Errori**

```
# cartella /geometria/
    # formule.py
    def area_rettangolo(base, altezza):
        return base * altezza
```

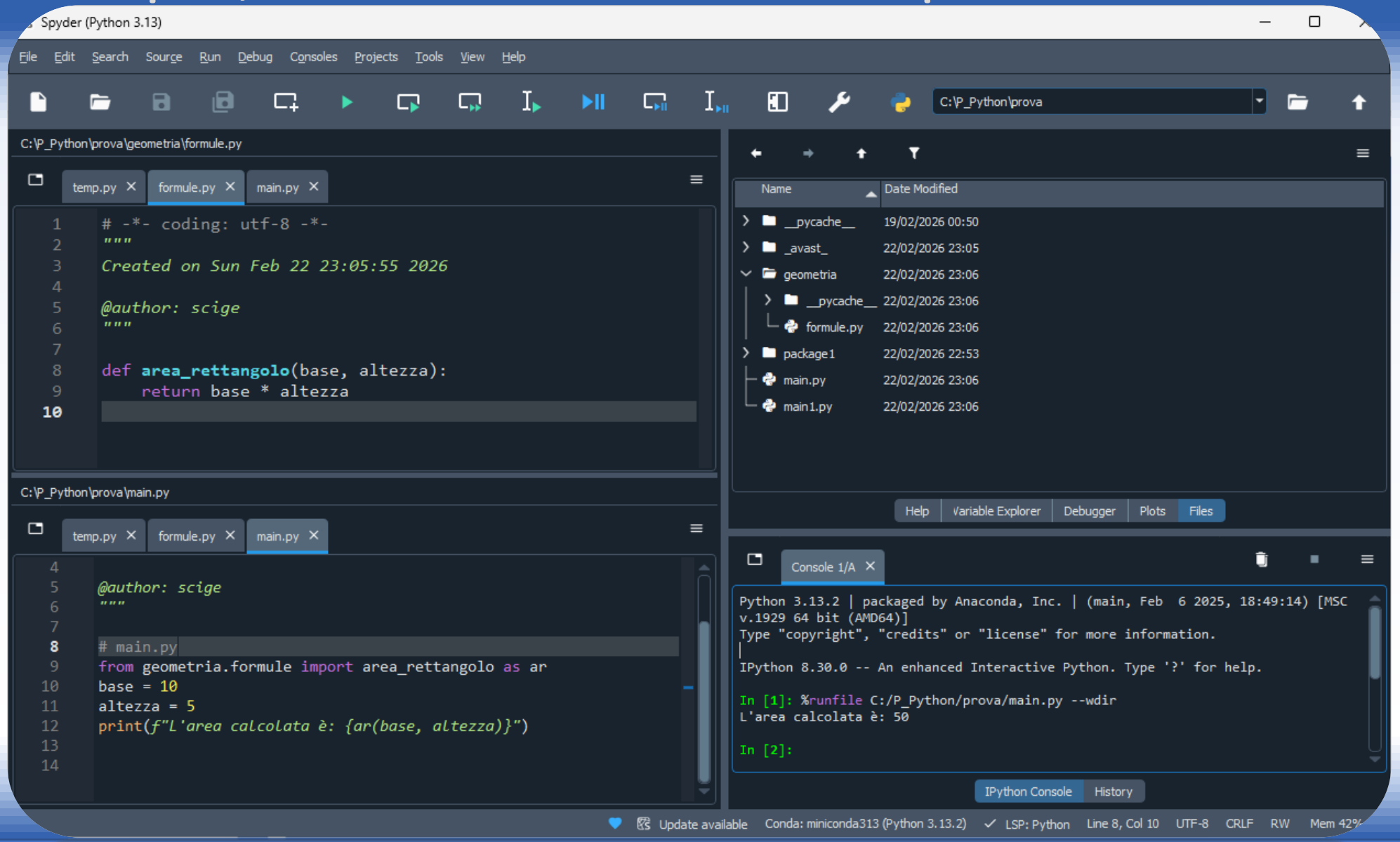
def FUNZIONE ():
pass

Passaggio ARGOMENTI

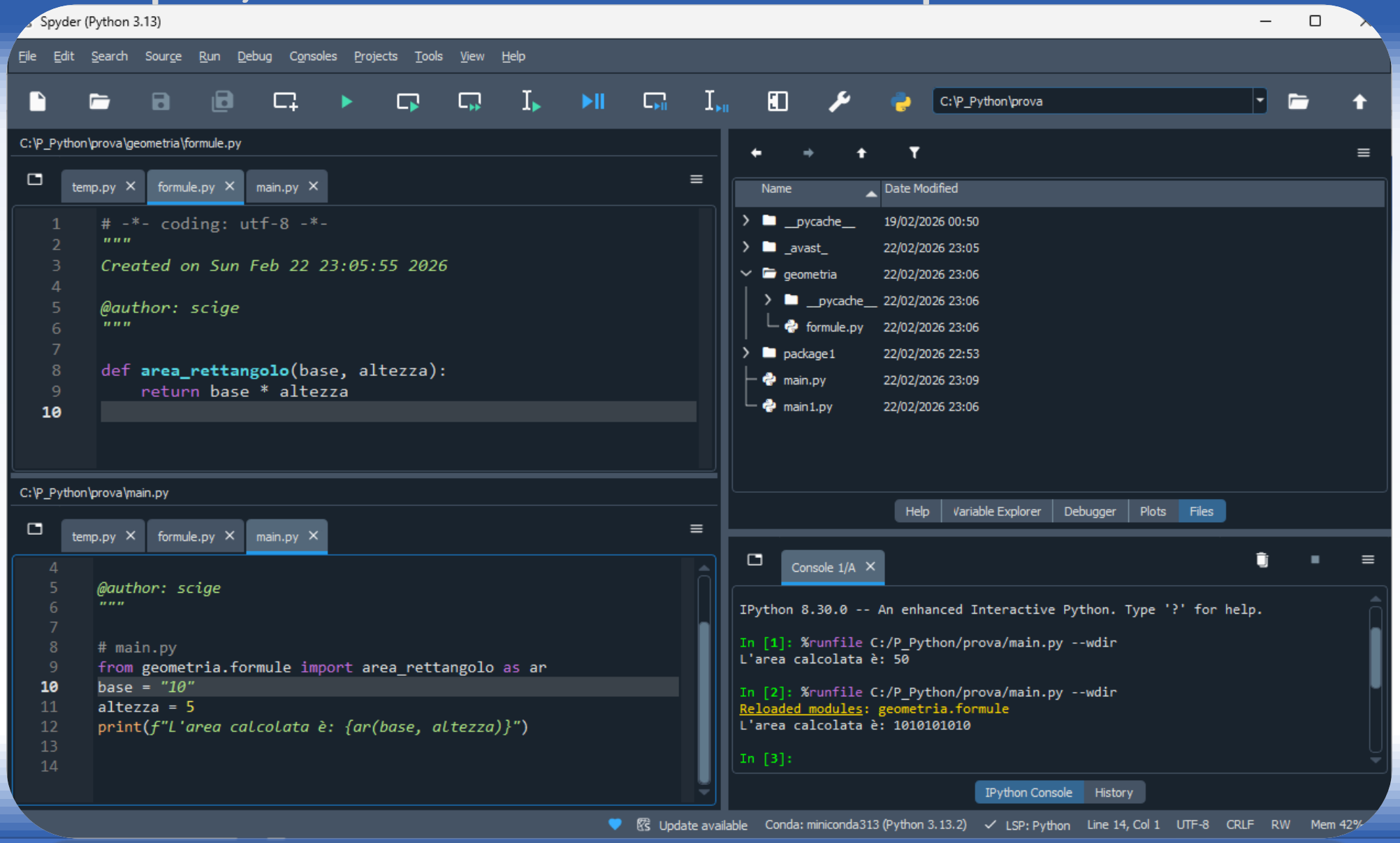
Valore di RITORNO

```
# cartella .
# main.py
from geometria.formule import area_rettangolo as ar
base = 10
altezza = 5
try:
    base = float(input("Inserisci base : "))
    altezza = float(input("Inserisci altezza : "))
except ValueError:
    print("Errore: devi inserire valori numerici!")
print(f"L'area calcolata è: {ar(base, altezza)}")
#
```

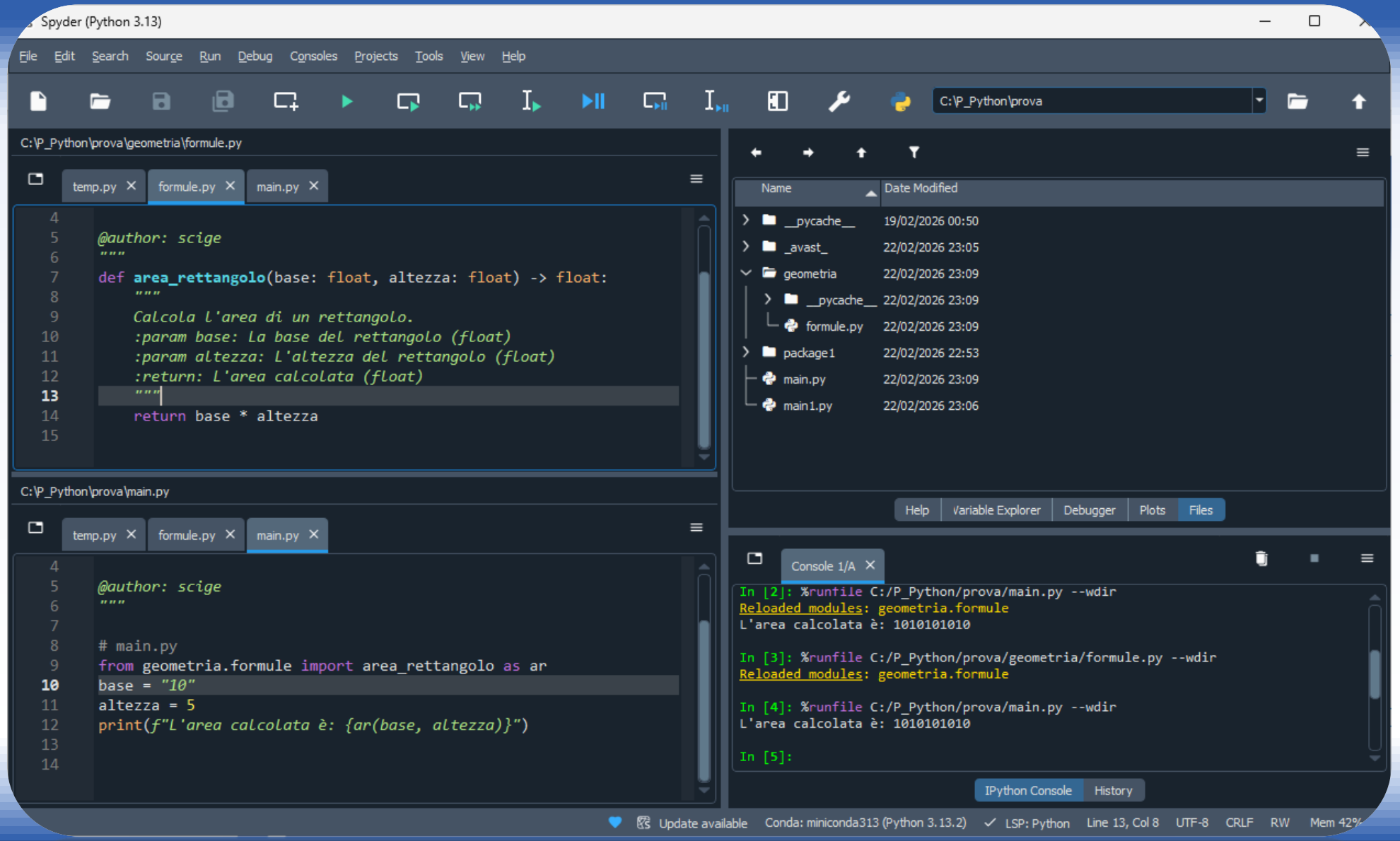
1.2. Corpus, Moduli e Cartelle - esempio



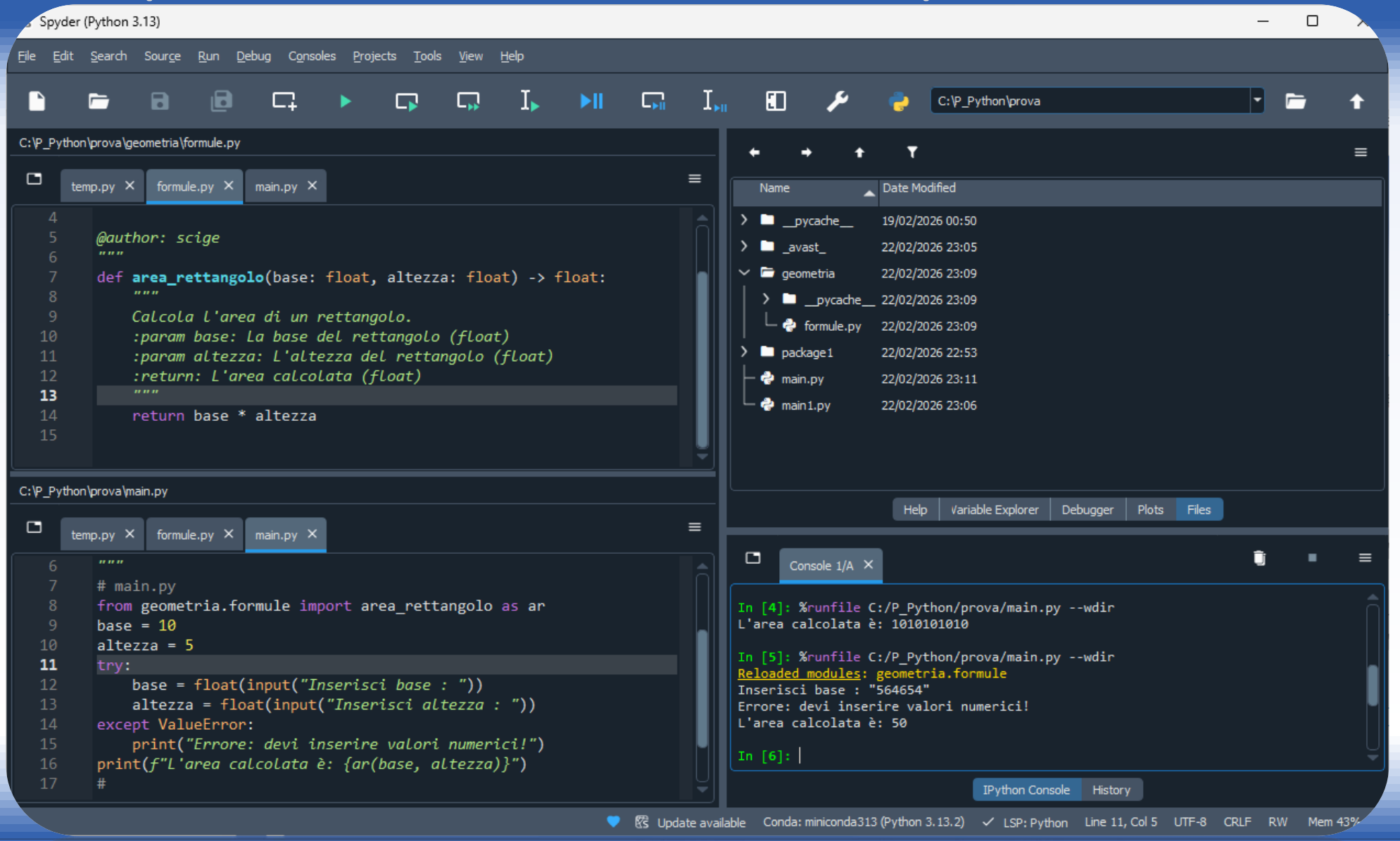
1.2. Corpus, Moduli e Cartelle - esempio



1.2. Corpus, Moduli e Cartelle - esempio

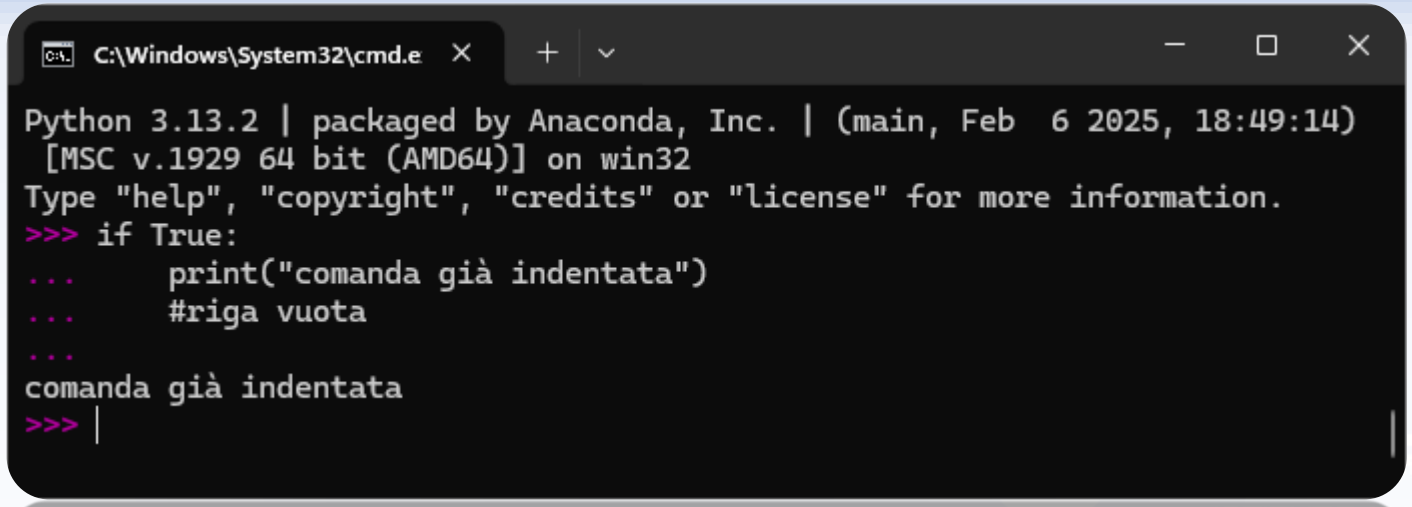


1.2. Corpus, Moduli e Cartelle - esempio

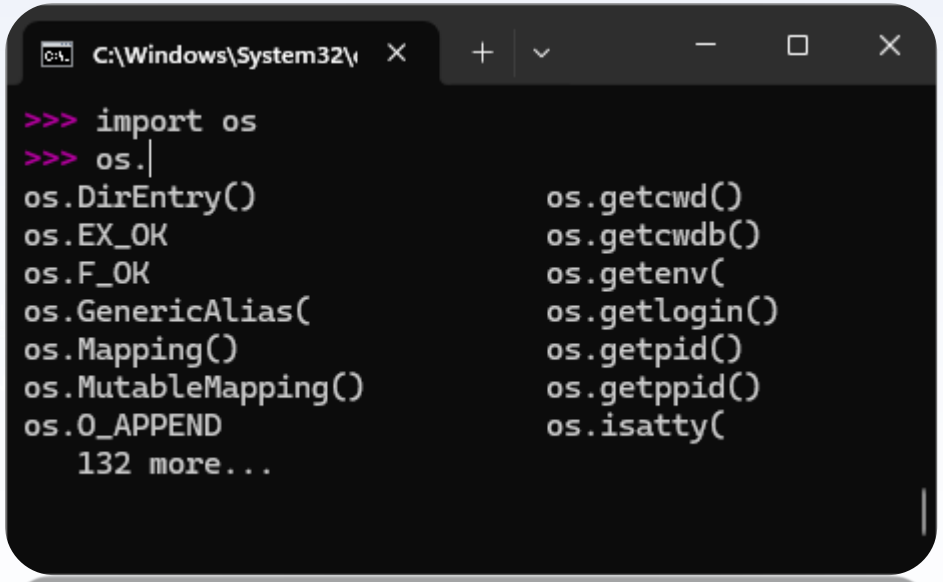


1.2. Corpus, Moduli e Cartelle - esempio

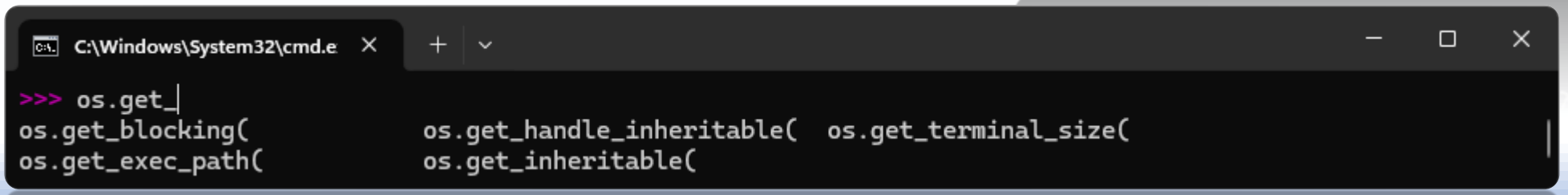
CORPUS IMMEDIATO : L'input immediato dell'interprete è assistito da interazioni di formattazioni



è possibile interagire tramite «discovery» [TAB+TAB]



La discovery implementa una «auto-complete»



1.2. Corpus, Moduli e Cartelle

CORPUS IN FILE – script codificato «stoccato» in files

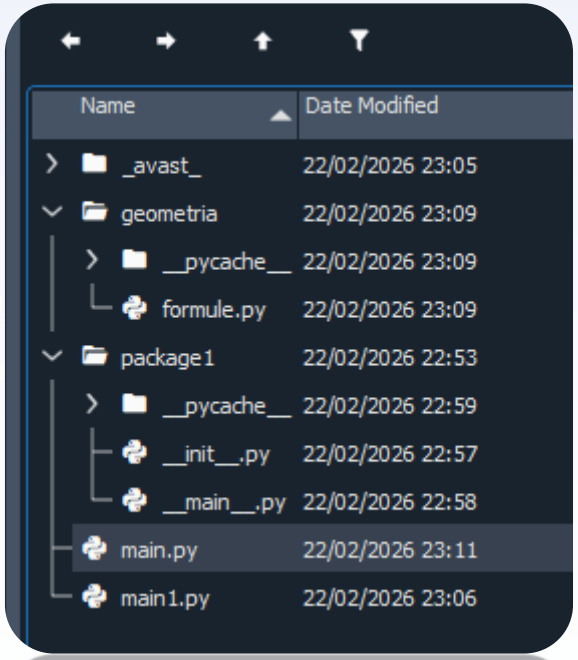
C:\P_Python\prova\geometria\formule.py

temp.py | formule.py | main.py

```python
@author: scige
"""
def area_rettangolo(base: float, altezza: float) -> float:
    """
    Calcola l'area di un rettangolo.
    :param base: La base del rettangolo (float)
    :param altezza: L'altezza del rettangolo (float)
    :return: L'area calcolata (float)
    """
    return base * altezza
```

←Il MODULO è sinonimo di «FILE» [standard C]

↓Le cartelle sono assimilate ai Packages




```
my_project/
├── main_app.py           # Top-level script
└── parent_package/       # The "Module" or Parent Package
    ├── __init__.py       # Initializer for parent
    └── sub_package/      # The Nested Package
        ├── __init__.py   # Initializer for sub-package
        └── __main__.py   # Entry point for sub-package
```

←La nidificazione di packages è per cartelle [tipo Java]

1.2. Corpus, Moduli e Cartelle – definizioni dinamiche

CORPUS DINAMICI – la definizione di funzioni/metodi è il risultato di interpretazioni del flusso di comandi in input.
Conseguenza → è possibile definire funzioni diverse in base all'input e quindi variegare l'effettivo comportamento

Definizione-In-Selettore ↓

```
mode = "advanced"

if mode == "advanced":
    def logic(x):
        return x ** 2
else:
    def logic(x):
        return x + 1

# Ora puoi chiamare 'logic' normalmente
print(logic(5)) # Output: 25
```

Definizione-In-Selettore ↓

```
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

use_triple = True

if use_triple:
    my_func = make_multiplier(3)
else:
    my_func = make_multiplier(2)

print(my_func(10)) # Output: 30
```

Definizione-In-Selettore ↓

```
use_cloud = True

if use_cloud:
    class Storage:
        def save(self, data):
            print("Salvataggio su AWS S3...")
else:
    class Storage:
        def save(self, data):
            print("Salvataggio su disco locale...")

# L'utente usa sempre 'Storage' senza sapere quale sia stata scelta
db = Storage()
db.save("test_data")
```

1.2. Corpus, Moduli e Cartelle - decoratori

DECORATORI sono strumenti più **eleganti**. Un decoratore è una funzione che "avvolge" un'altra per modificarne il comportamento, non cambiare il codice originale. La funzione risultante risulta con funzionalità aggiuntive.

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 22 23:34:21 2026

@author: scige
"""

from functools import wraps

# 1. Definiamo il decoratore che accetta il parametro N
def ripeti(n):
    def decoratore(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Eseguiamo la funzione originale N volte
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decoratore

# 2. Usiamo il decoratore con la sintassi @
@ripeti(5)
def mioPrint(testo):
    print(testo)

# 3. Test dell'esecuzione
mioPrint("Ciao! Questa riga apparirà 5 volte.")
```