APL64: Using ⎕PY - Python engine

Contents

Summary

The APL64 ⎕PY system function can create an instance of the Python engine within the current APL64 interpreter instance. The Python engine created using ⎕PY persists during the entire APL64 instance. When an APL64 instance is closed, the Python engine, if any, is disposed.

The □PY Exec action can execute a Python script. The □PY Eval action, with synonym □PY Get, can obtain the value of a Python expression. The □PY Set action can set a Python variable value with an APL 64 value.

The □PY Exec, Eval, Get and Set actions support value substitution, so that APL variable values can be used a Python script or expression.

The interface between Python and .Net is implemented by the PythonNet Nuget package included in the APL64 interpreter. The installed version of Python must be compatible with the PythonNet interface.

The □PY implementation is cross-platform compatible.

Prerequisites

It is up to the APL64 programmer or end user of an APL64-based application to assure that an appropriate version of no-cost Python is installed on the target workstation, e.g. Python v3.14+.

The path of the Python dll must be provided in the right argument of the □PY InitPy action to create the Python engine in an APL64 instance.

Multiple instances of the Python engine cannot be created within an APL64 interpreter instance. This is a Python-imposed restriction. The Python engine supports multiple scopes, so each □WSE engine and its APL instance parent have separate scopes.

Python does not support character scalars or arrays. For the best □PY experience convert APL character data to APL string data before sending the data to the Python engine. Character data set to the Python engine are converted to string data and returned to APL as string data.

□PY Actions

Syntax: [result] ← □PY Action actionArg1 ...

The □PY action text is not case sensitive.

EVAL (synonym: GET)

The EVAL action runs the evalExpr in the Python engine and returns the result.

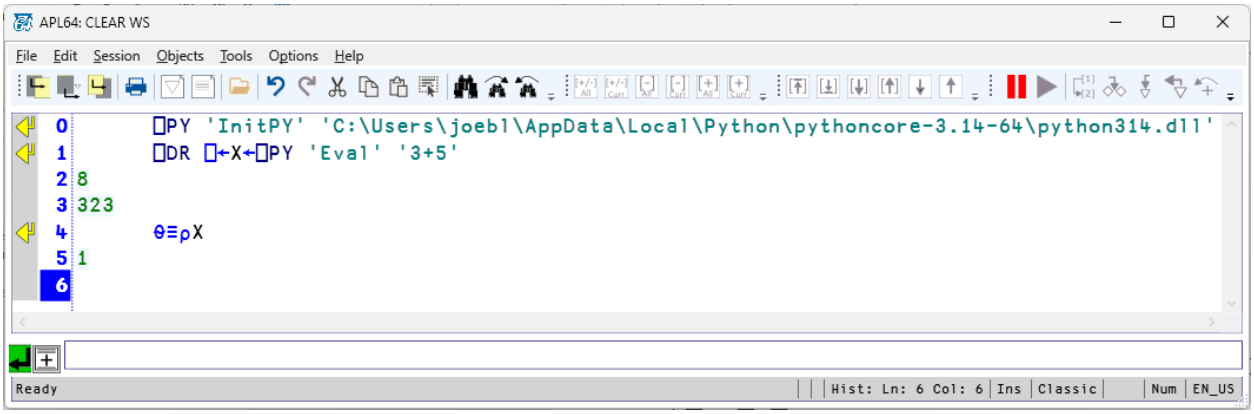
Syntax: [result] ← □PY 'Eval' evalExpr [subVal0 subVal1 ...]

evalExpr is the text Python expression to evaluate. The evalExpr can contain substitution indicators of the for {0}{1} ...

subVal1 ... are optional value substitution arguments.

Example: Arithmetic operations

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕DR ⎕←X←⎕PY 'Eval' '3+5'
⍬≡⍴X
```



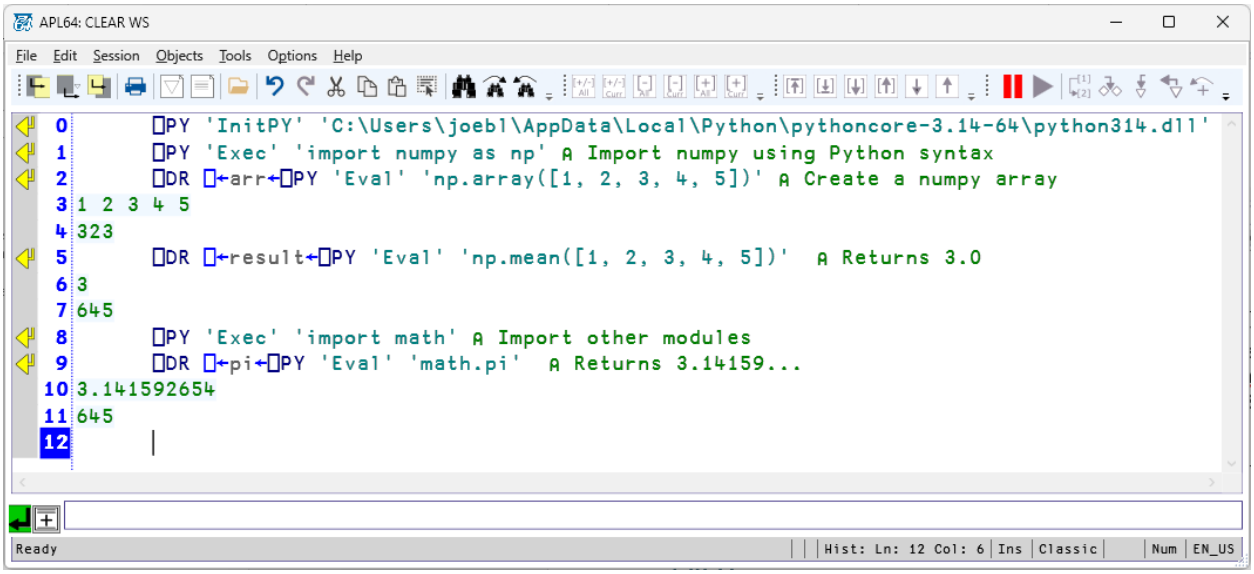
Example: Using Python modules

This example uses the Python ‘numpy’ module which can be installed using the Windows Command Prompt (as administrator) with this command:

```
py -m pip install numpy
```

If the ‘numpy’ module is installed, run this Python script:

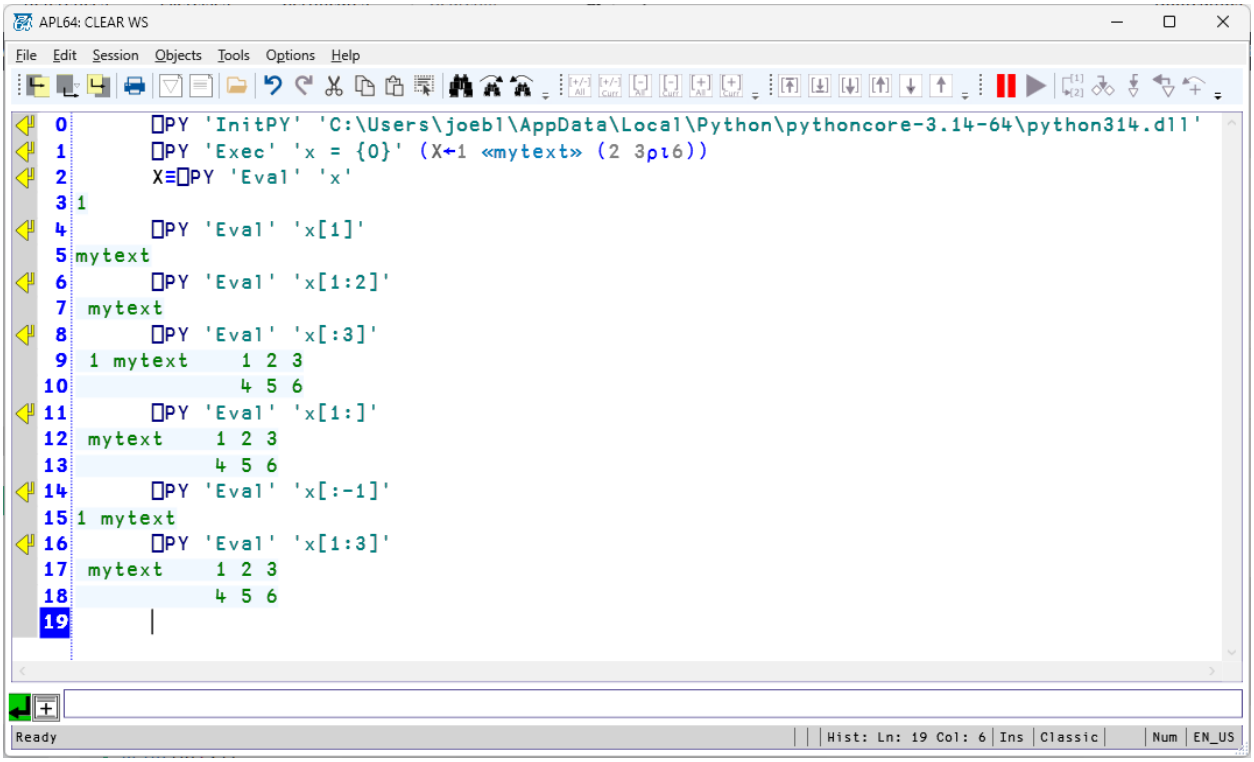
```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Exec' 'import numpy as np' ⍝ Import numpy using Python syntax
⎕DR ⎕←arr←⎕PY 'Eval' 'np.array([1, 2, 3, 4, 5])' ⍝ Get a numpy array
⎕DR ⎕←result←⎕PY 'Eval' 'np.mean([1, 2, 3, 4, 5])'  ⍝ Returns 3.0
⎕PY 'Exec' 'import math' ⍝ Import other modules
⎕DR ⎕←pi←⎕PY 'Eval' 'math.pi'  ⍝ Returns 3.14159...
```



Example: Slicing an array

Use caution when slicing an array, because for certain array types the original array may be affected.

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Exec' 'x = {0}' (X←1 «mytext» (2 3⍴⍳6))
X≡⎕PY 'Eval' 'x'
⎕PY 'Eval' 'x[1]'
⎕PY 'Eval' 'x[1:2]'
⎕PY 'Eval' 'x[:3]'
⎕PY 'Eval' 'x[1:]'
⎕PY 'Eval' 'x[:-1]'
⎕PY 'Eval' 'x[1:3]'
```



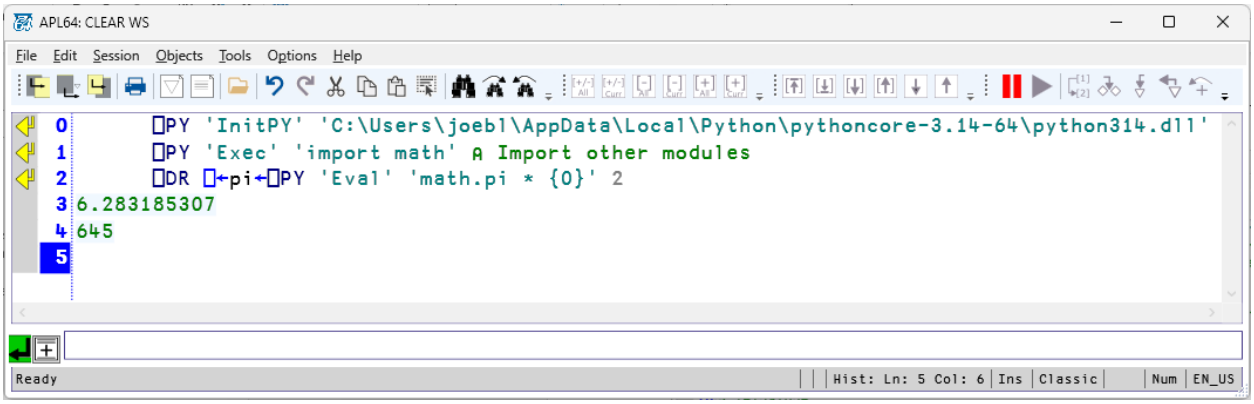
Value Substitution

The evalExpr can include one or more value substitution indicators, {0}, {1}, ... A value substitution indicator with a specific index can be included more than once in an evalExpr. There must be one APL substitution value for each value substitution indicator index.

Value substitution occurs before the statement is evaluated in the Python engine.

Example:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Exec' 'import math' ⍝ Import other modules
⎕DR ⎕←pi←⎕PY 'Eval' 'math.pi * {0}' 2
```



EXEC

The EXEC action runs the

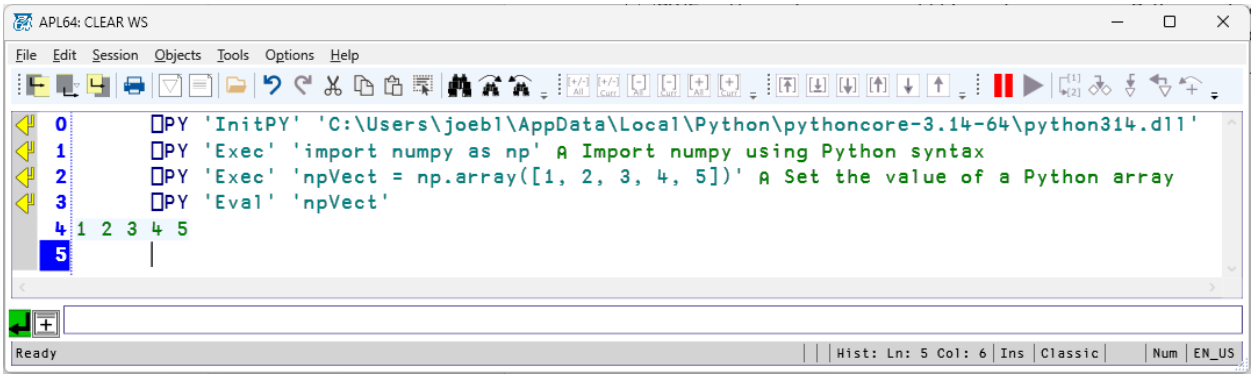
Syntax: ⎕PY 'Exec' 'pythonScript' [subVal0 subVal1 ...]

Example: Single line script

Single Line Scripts

Example:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Exec' 'import numpy as np' ⍝ Import numpy using Python syntax
⎕PY 'Exec' 'npVect = np.array([1, 2, 3, 4, 5])' ⍝ Set the value of a Python array
⎕PY 'Eval' 'npVect'
```

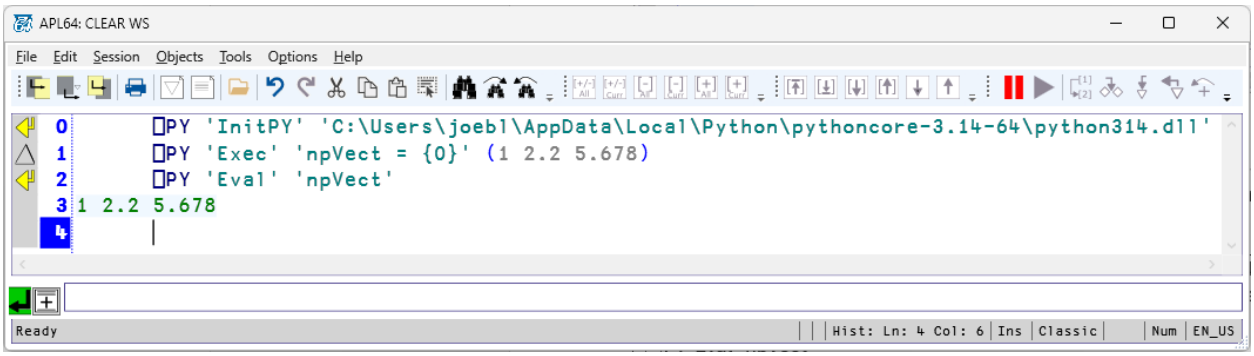


Value Substitution into a Python script

The evalExpr can include one or more value substitution indicators, {0}, {1}, ... A value substitution indicator with a specific index can be included more than once in an evalExpr. There must be one APL substitution value for each value substitution indicator index.

Value substitution occurs before the script is executed in the Python engine.

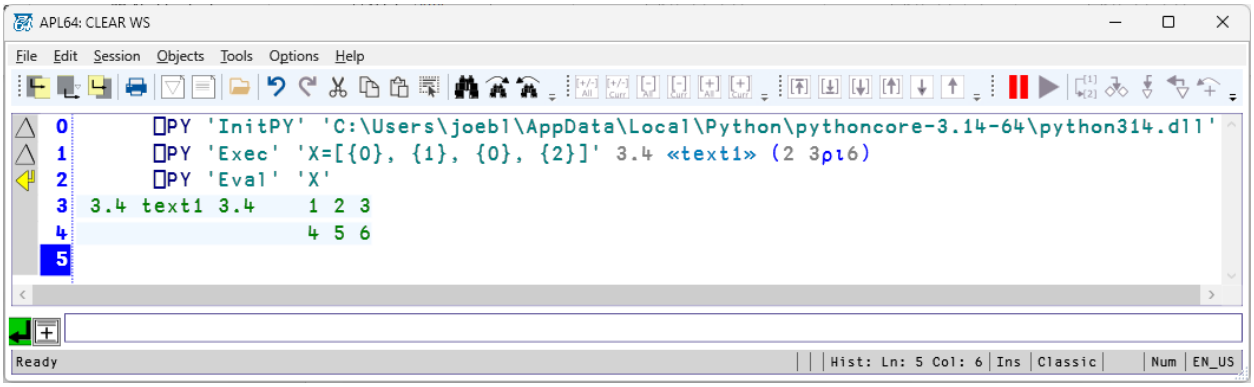
```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Exec' 'npVect = {0}' (1 2.2 5.678)
⎕PY 'Eval' 'npVect'
```



Example: Value substitution multiple mixed data types

In this example the substituted values have different data types. The {0} substitution occurs two times in the Python script.

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
```



Multi-line Python Scripts

A multi-line Python script is a text vector which contains new line characters separating the script lines. Python comments can be included in the script with the prefix '#'.

When a Python script is created in an APL64 programmer-defined function, the APL64 string line continuation structure is convenient: S←«« ... »» or S←« ... ».

Example: Multi-line Python script via Line Continuation: Create and use a Python function:

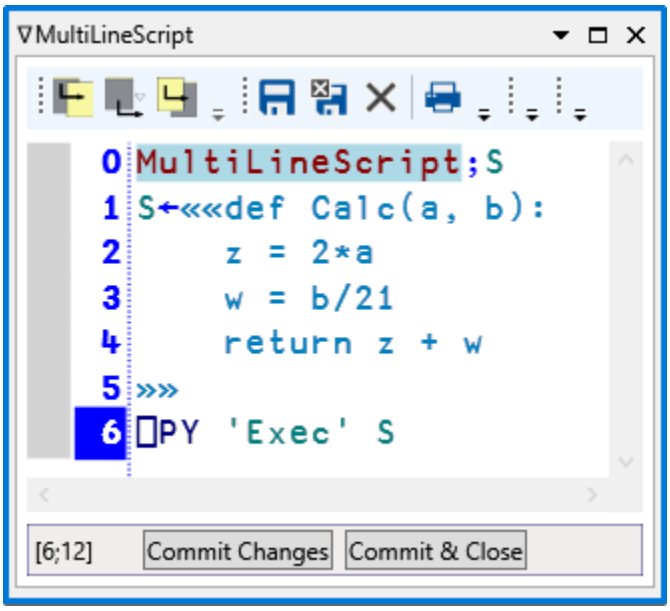
In this example a multi-line Python script is created in an APL64 programmer defined function. The script is created using the S←«« ... »» line continuation structure. The function runs the Python script using ⎕PY 'Exec' S. The Python script defines a Python function, which requires Python-specific indentation within the function definition.

First initialize the Python engine in the APL64 instance:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
```

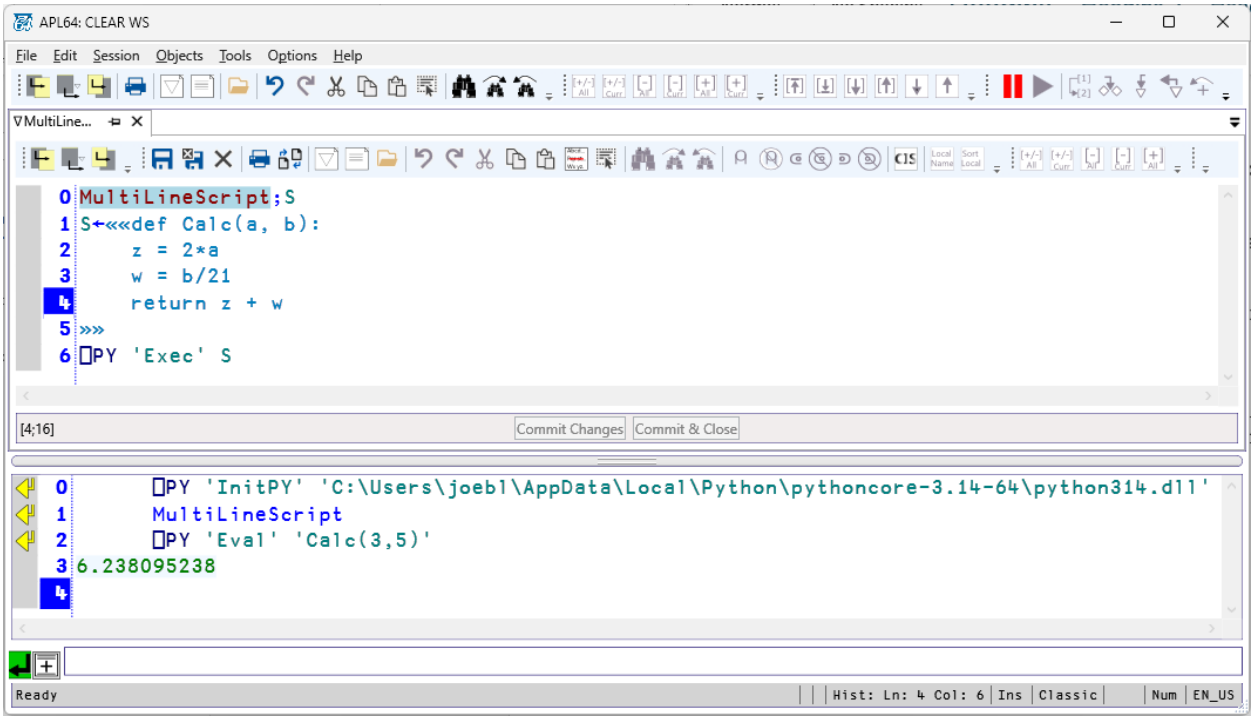
Create and run the MultiLineScript function to create the Python function:

```
MultiLineScript;S
S←⍬⍬def Calc(a, b):
   z = 2*a
   w = b/21
   return z + w
⍬⍬⍬
⎕PY 'Exec' S
```



Use the ⎕PY 'Eval' action to obtain a result:

```
⎕←result ← ⎕PY 'Eval' 'Calc(3, 5)'
```



Example: Multi-line Python script with value substitution

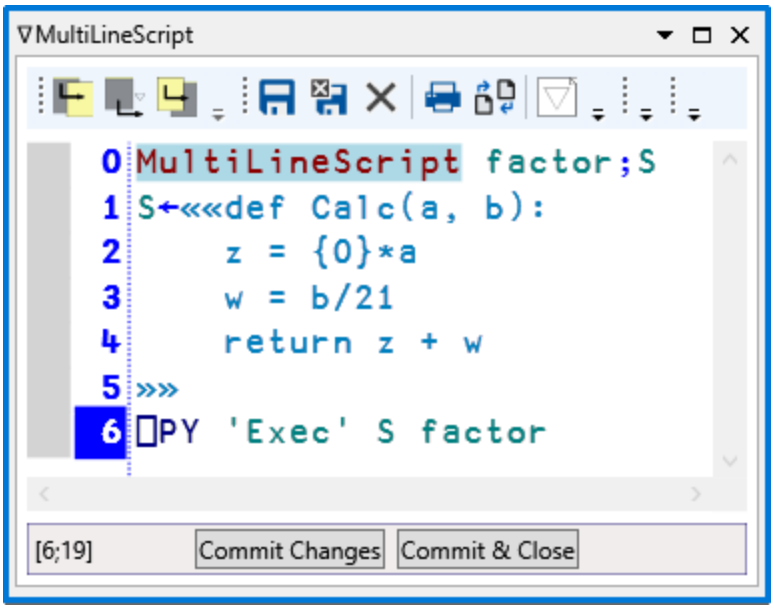
In this example the APL64 programmer-defined function has a right argument as a value substitution parameter in the Python script.

First initialize the Python engine in the APL64 instance:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
```

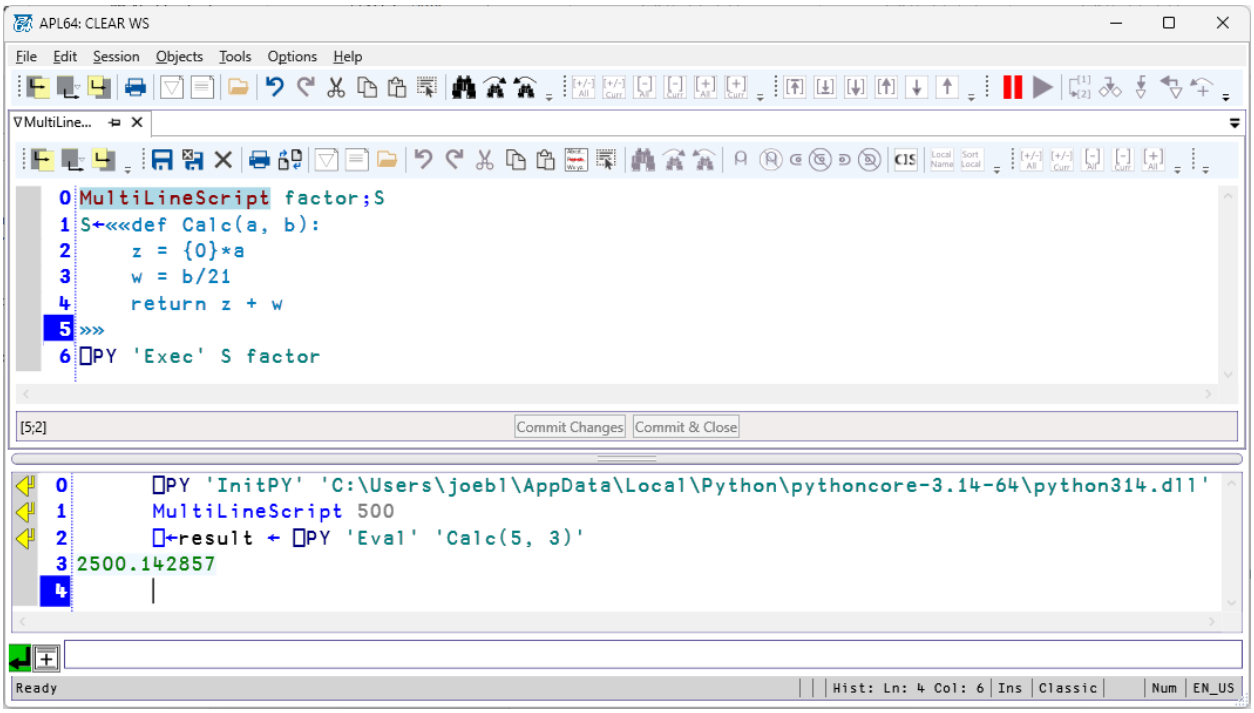
Create and run the MultiLineScript function to create the Python function:

```
MultiLineScript factor;S
S←⍬⍬def Calc(a, b):
  z = {0}*a
  w = b/21
  return z + w
⍬⍬⍬
⎕PY 'Exec' S factor
```



Use the ⎕PY 'Eval' action to obtain a result:

```
⎕←result ← ⎕PY 'Eval' 'Calc(5, 3)'
```



APL: Transfer Control between APL and Python within a Python script

Within a script it is possible to transfer execution control between the APL64 interpreter and the Python engine. To execute a script line in the APL64 interpreter, prefix the APL64 executable statement on that line with 'APL:'. The Python portions of the script must be

consistent and complete within themselves. Value substation into the script applies only to the Python portions of the script.

Example:

In this example there are three portions of the script: Python sub-script 1, APL statement, Python sub-script 2. This script also uses value substation on the Python portions of the script.

Initialize the Python engine:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
```

Create and run the PyScript function:

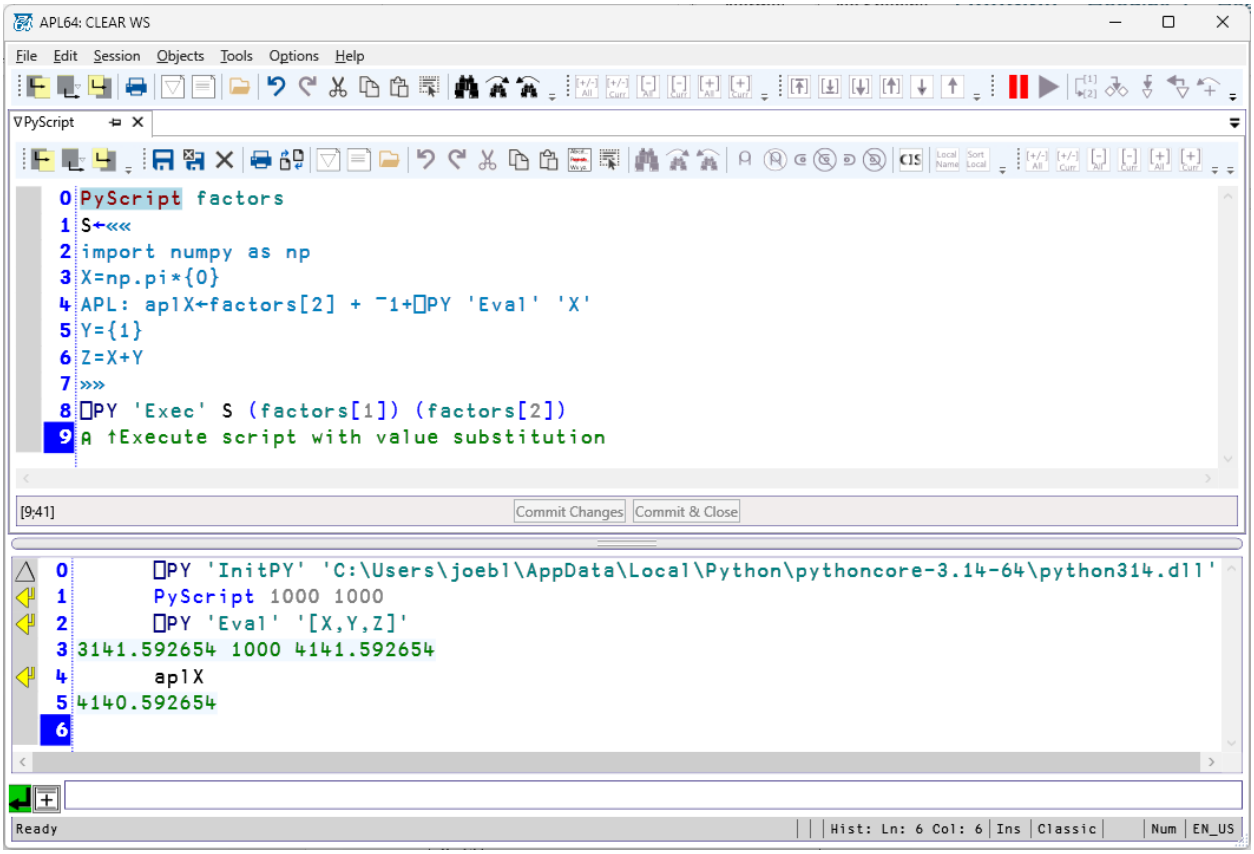
```
PyScript factors
S←««
import numpy as np
X=np.pi*{0}
APL: aplX←factors[2] + ¯1+⎕PY 'Eval' 'X'
Y={1}
Z=X+Y
»»
⎕PY 'Exec' S (factors[1]) (factors[2])
⍝ ↑Execute script with value substitution
```

Use the ⎕PY 'Eval' action to obtain the values of Python variables defined in the script:

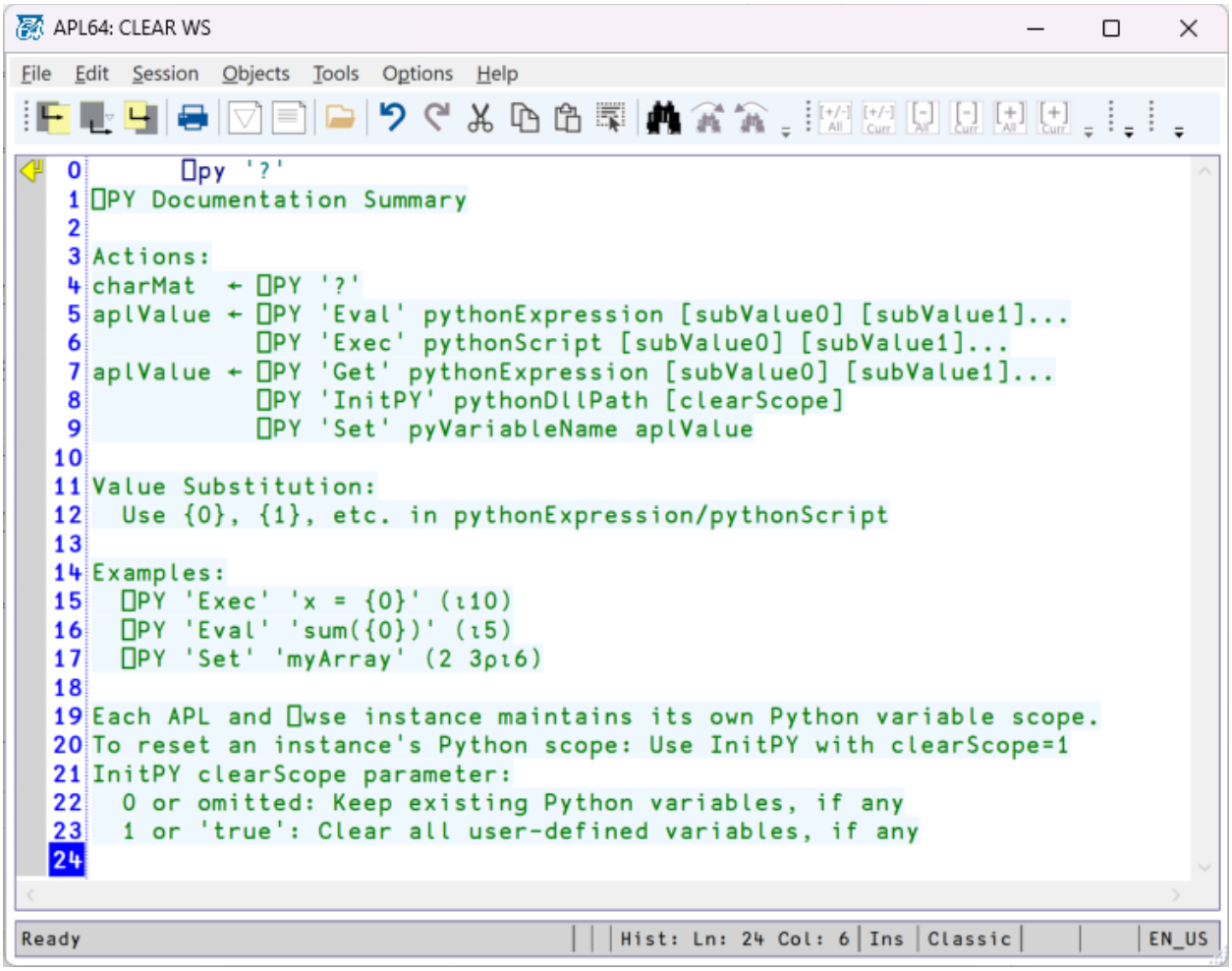
```
⎕PY 'Eval' '[X,Y,Z]'
```

Check the value of the APL variable created in the APL portion of the script:

```
aplX
```



HELP (?)



INITPY

Syntax:]PY 'InitPY' pythonDllPath [clearScope]

The InitPy action must be used before any other]PY actions are used. The required right argument, pythonDllPath, is the path to the Python dll installed on the target workstation.

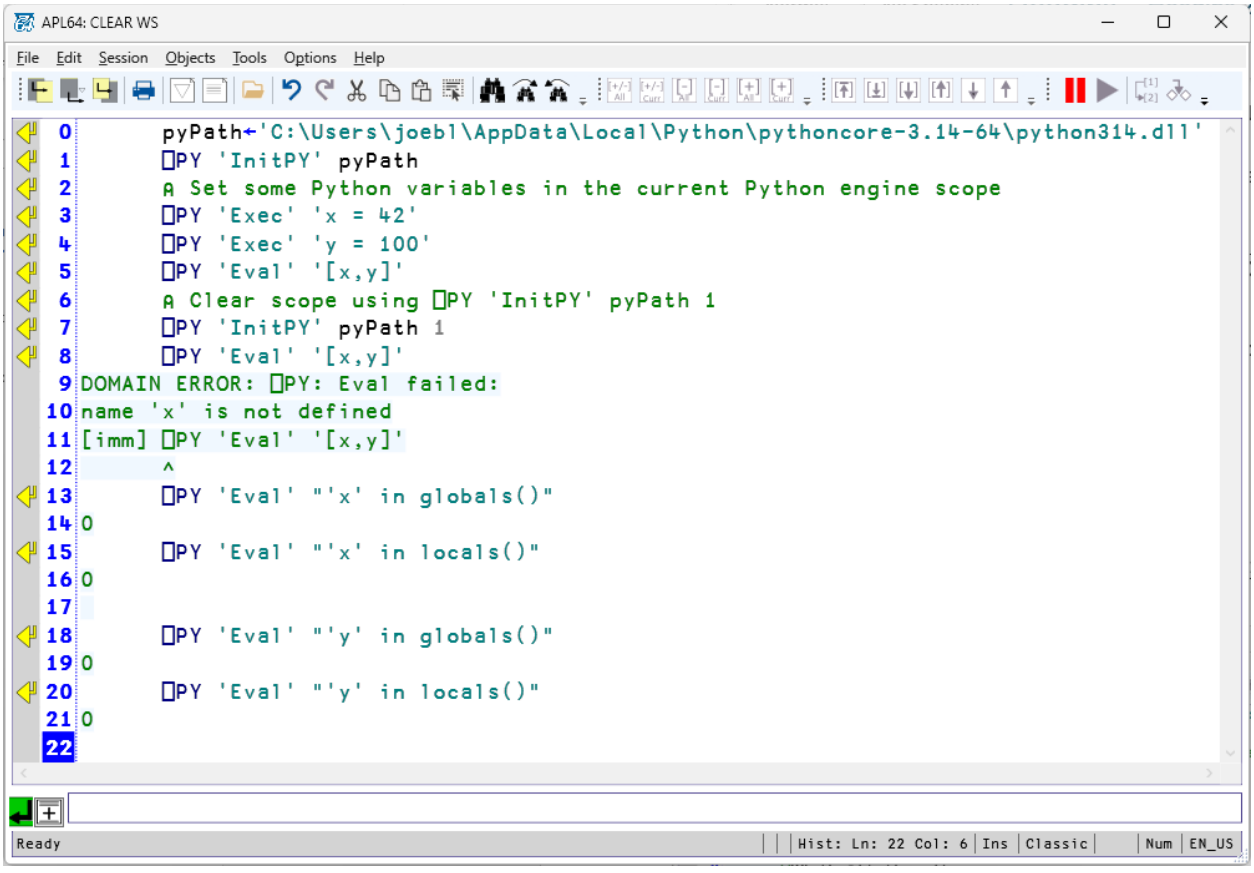
The optional argument, clearScope, determines if the current Python engine scope will be initialized. The current scope contains the Python modules and variables which have been created in the current APL64 instance. The Python engine supports multiple scopes.

clearScope: 0/Do not clear scope, 1/Clear scope

Multiple Python engine scopes occur in an APL64 instance only if the parent APL64 instance creates one or more]WSE workspace engines children. In this scenario the parent APL instance and each]WSE instance have distinct and separate Python engine scopes.

Example:

```
pyPath←'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'InitPY' pyPath
⍝ Set some Python variables in the current Python engine scope
⎕PY 'Exec' 'x = 42'
⎕PY 'Exec' 'y = 100'
⎕PY 'Eval' '[x,y]'
⍝ Clear scope using ⎕PY 'InitPY' pyPath 1
⎕PY 'InitPY' pyPath 1
⎕PY 'Eval' '[x,y]'
⎕PY 'Eval' "'x' in globals()"
⎕PY 'Eval' "'x' in locals()"
⎕PY 'Eval' "'y' in globals()"
⎕PY 'Eval' "'y' in locals()"
```




```
⎕PY 'Exec' 'x = {0}' (1 'text' (2 3))      ⍝ Mixed with nested
⎕PY 'Exec' 'y = {0}' (1 2.3 'hello' (4 5 6))  ⍝ Multiple types
⎕PY 'Eval' 'x[1]'
```

```
⍝ Set array variable
⎕PY 'Exec' 'X={0}' (⍳10)          ⍝ X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

⍝ Use array in expression
⎕PY 'Eval' 'sum({0})' (⍳5)        ⍝ Returns sum of array

⍝ Matrix becomes nested list
⎕PY 'Exec' 'M={0}' (2 3⍴⍳6)       ⍝ M = [[1, 2, 3], [4, 5, 6]]

⍝ Set action
⎕PY 'Set' 'myArray' (⍳10)         ⍝ myArray = [1, 2, 3, ...]

⍝ String with special characters
⎕PY 'Exec' 's={0}' 'Hello "World"' ⍝ s = "Hello \"World\""
```

SET

Syntax: ⎕PY 'Set' pyVariableName aplValue

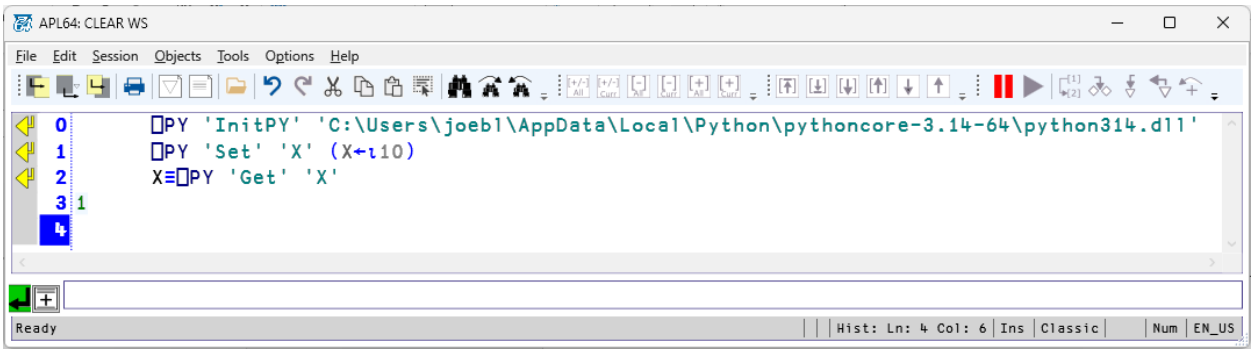
pyVariableName is an APL64 text value which is suitable for a Python variable name. The named Python variable will be created if it does not exist.

aplValue is an APL64 value which is suitable for a Python variable value.

Python ultimately sets variable values using the Python text representation of the data. APL values are converted to their Python text representation when setting the value of a Python variable. For APL64 floating point data, the maximum available precision is used to create the Python text representation.

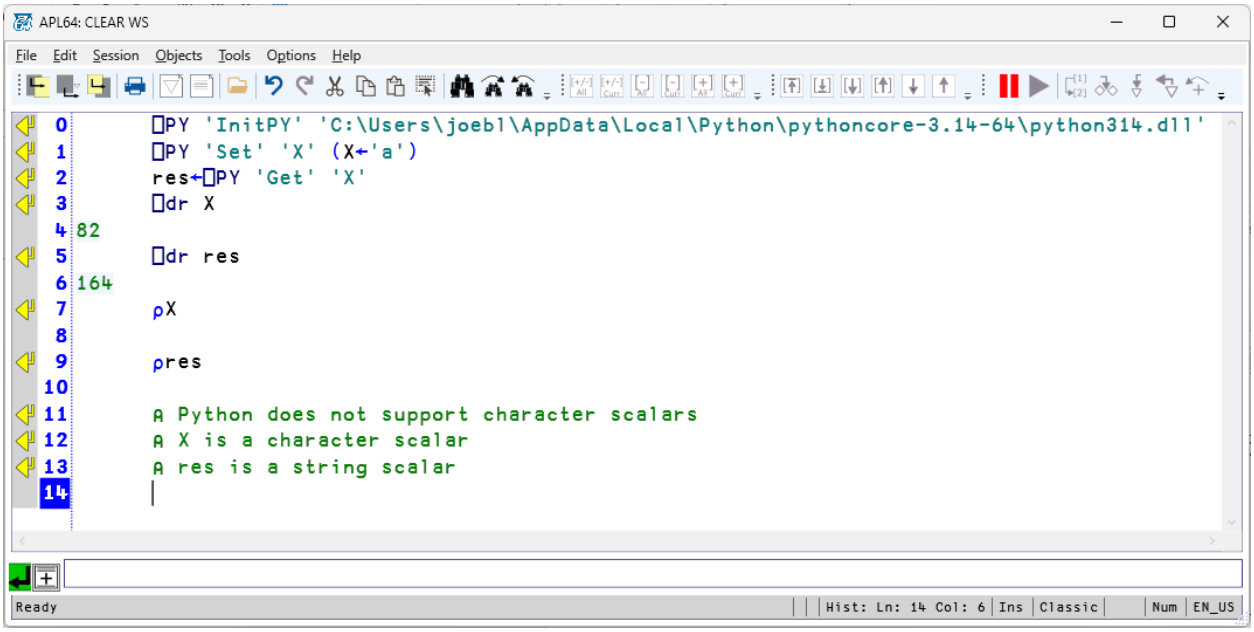
Example:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Set' 'X' (X←⍳10)
X≡⎕PY 'Get' 'X'
```



Example

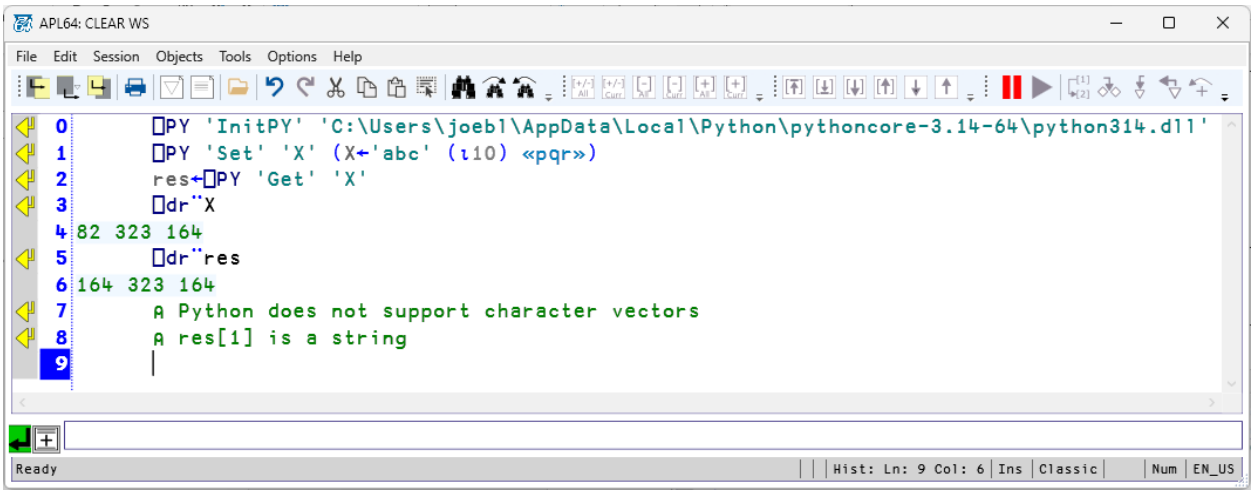
```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Set' 'X' (X←'a')
res←⎕PY 'Get' 'X'
⎕dr X
⎕dr res
⍝ Python does not support character scalars
⍝ res[1] is a string
```



Example:

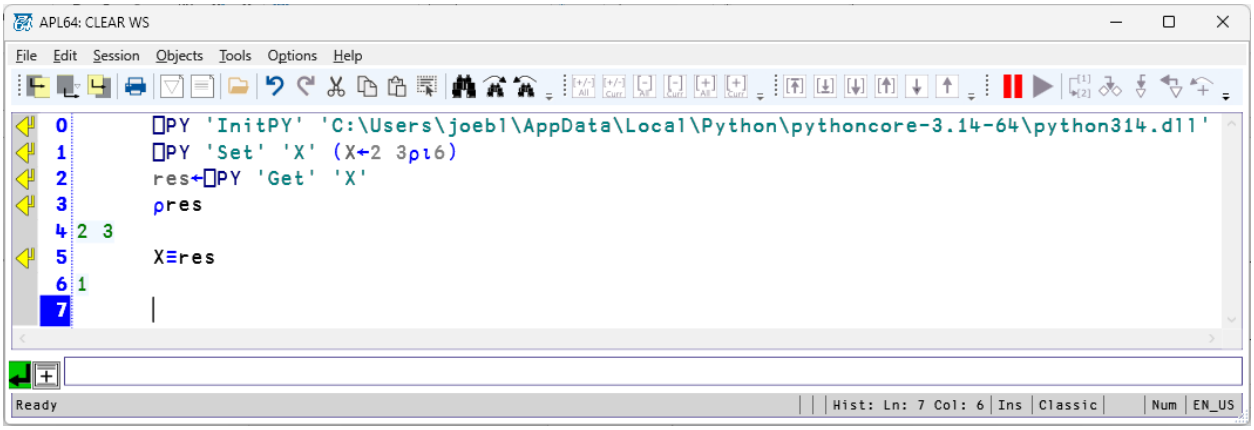
```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Set' 'X' (X←'abc' (⍳10) «pqr»)
res←⎕PY 'Get' 'X'
⎕dr¨X
⎕dr¨res
```

⍝ Python does not support character vectors
⍝ res[1] is a string



Example:

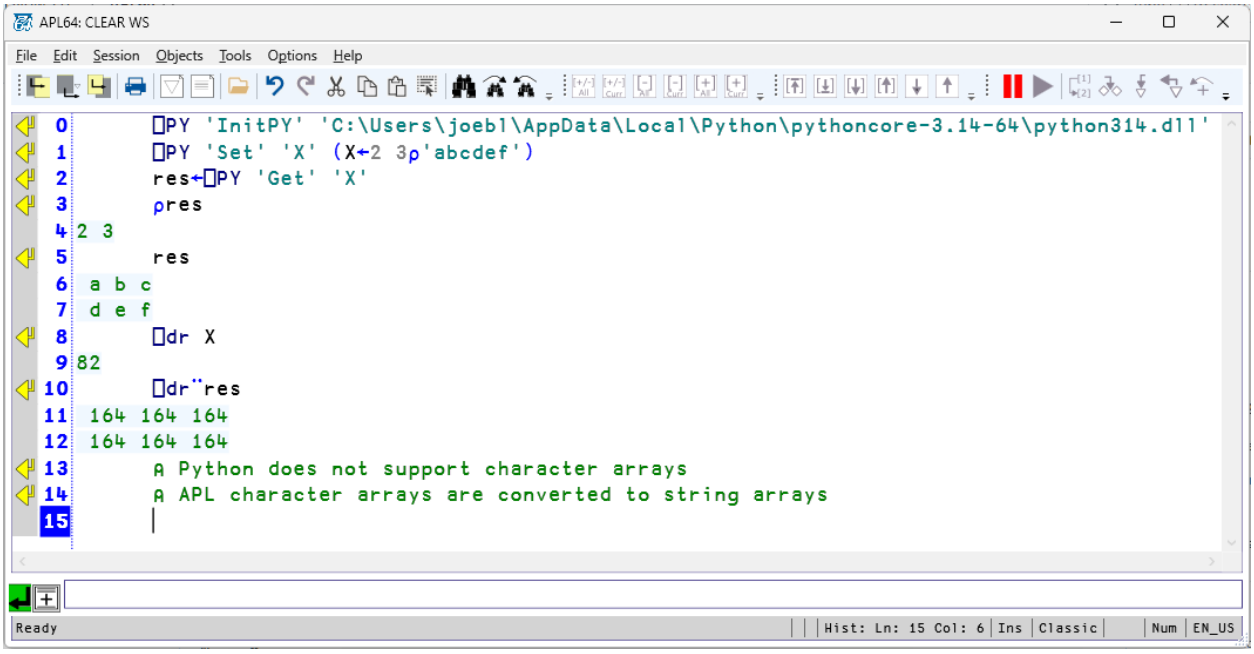
```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Set' 'X' (X←2 3⍴⍳6)
res←⎕PY 'Get' 'X'
X≡res
⍴res
```



Example:

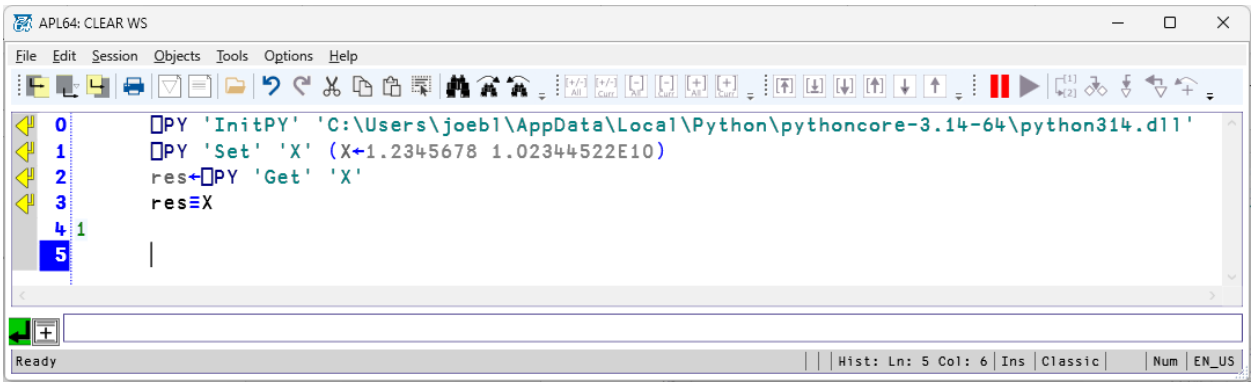
```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Set' 'X' (X←2 3⍴'abcdef')
res←⎕PY 'Get' 'X'
⍴res
⎕dr X
```

```
⎕dr¨res
⍝ Python does not support character arrays
⍝ APL character arrays are converted to string arrays
```



Example:

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Set' 'X' (X←1.2345678 1.02344522E10)
res←⎕PY 'Get' 'X'
res≡X
```

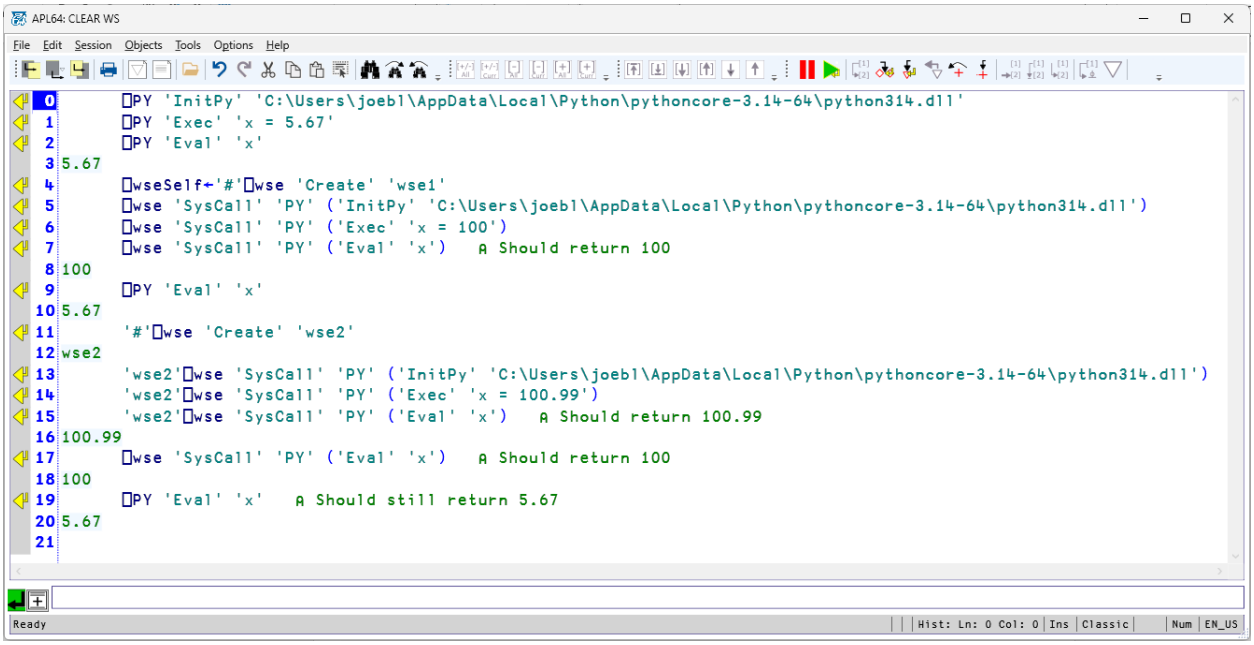


□PY and □WSE

□PY can be used in a □WSE instance. Since a □WSE instance is an in-process server, the same Python engine is shared by the parent APL64 instance and all □WSE child instances. If an APL64 instance creates one or more □WSE workspace engines, the Python instance which may be created by the parent APL64 instance and the single Python instance which may be created by a □WSE workspace engine are separate and distinct Python instances.

Example: One parent APL instance and two □WSE child instances.

```
pyPath←'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
□PY 'InitPy' pyPath
□PY 'Exec' 'x = 5.67'
□PY 'Eval' 'x'
□wseSelf←'#'□wse 'Create' 'wse1'
□wse 'SysCall' 'PY' ('InitPy' pyPath ')
□wse 'SysCall' 'PY' ('Exec' 'x = 100')
□wse 'SysCall' 'PY' ('Eval' 'x')  ⍝ Should return 100
□PY 'Eval' 'x'
'#'□wse 'Create' 'wse2'
'wse2'□wse 'SysCall' 'PY' ('InitPy' pyPath)
'wse2'□wse 'SysCall' 'PY' ('Exec' 'x = 100.99')
'wse2'□wse 'SysCall' 'PY' ('Eval' 'x')  ⍝ Should return 100.99
□wse 'SysCall' 'PY' ('Eval' 'x')  ⍝ Should return 100
□PY 'Eval' 'x'  ⍝ Should still return 5.67
```



□PY and □HTLC/□HTLS

□PY can be used in an □HTLC client and in an □HTLS server. An □HTLS server is an out of process server, so the client and the server will have different Python engine instances with their respective Python engine scopes.

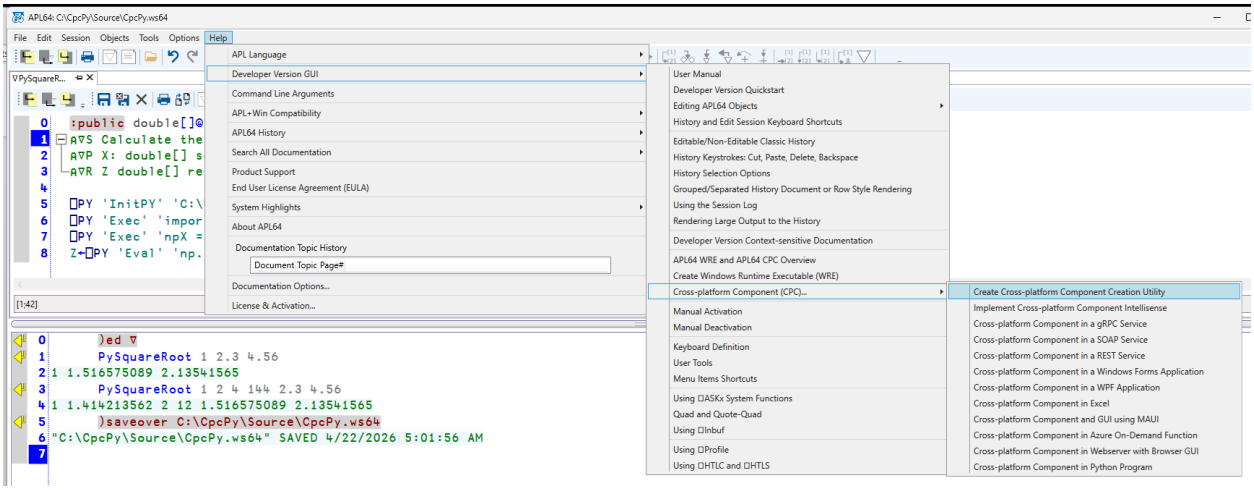
An APL64 instance which is an □HTLC client may create one Python instance. An APL64 instance which is an □HTLS server can create one Python instance. When an APL64 instance is an □HTLC client and uses the 'CallAsync' action, the APL64 instance also acts as an □HTLS server to receive and handle the asynchronous response, but that □HTLC instance can create only one Python instance.

□PY and APL64 Runtime

□PY can be used in an APL64 windows runtime executable (WRE) and in an APL64 cross-platform component (CPC).

Example: Using □PY in an APL64 cross-platform component (CPC)

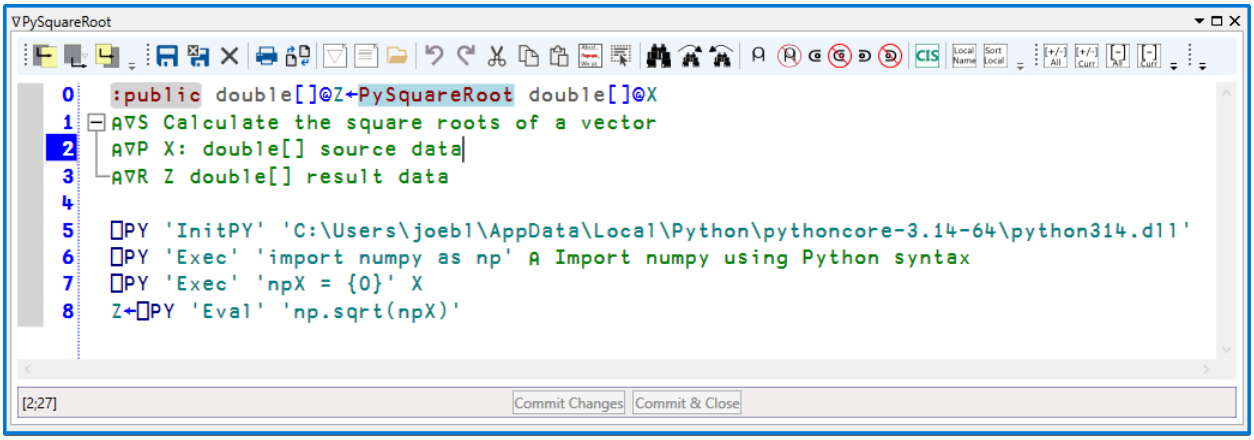
This example assumes that the reader has basic familiarity with the APL64 CPC creation utility and elementary C# programming. For more information refer to this documentation:



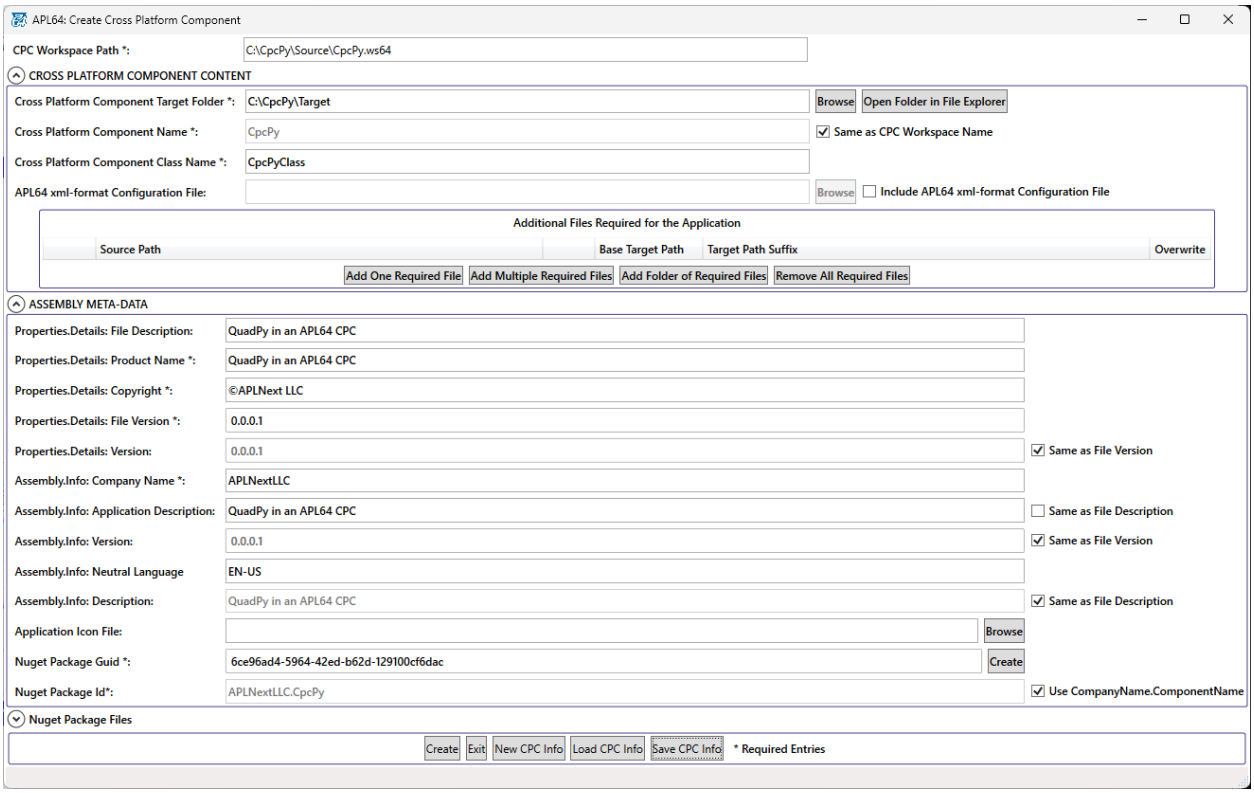
Sample public function:

```
:public double[]@Z←PySquareRoot double[]@X
⍝∇S Calculate the square roots of a vector
⍝∇P X: double[] source data
⍝∇R Z double[] result data
```

```
⎕PY 'InitPY' 'C:\Users\joebl\AppData\Local\Python\pythoncore-3.14-64\python314.dll'
⎕PY 'Exec' 'import numpy as np' ⍝ Import numpy using Python syntax
⎕PY 'Exec' 'npX = {0}' X
Z←⎕PY 'Eval' 'np.sqrt(npX)'
```



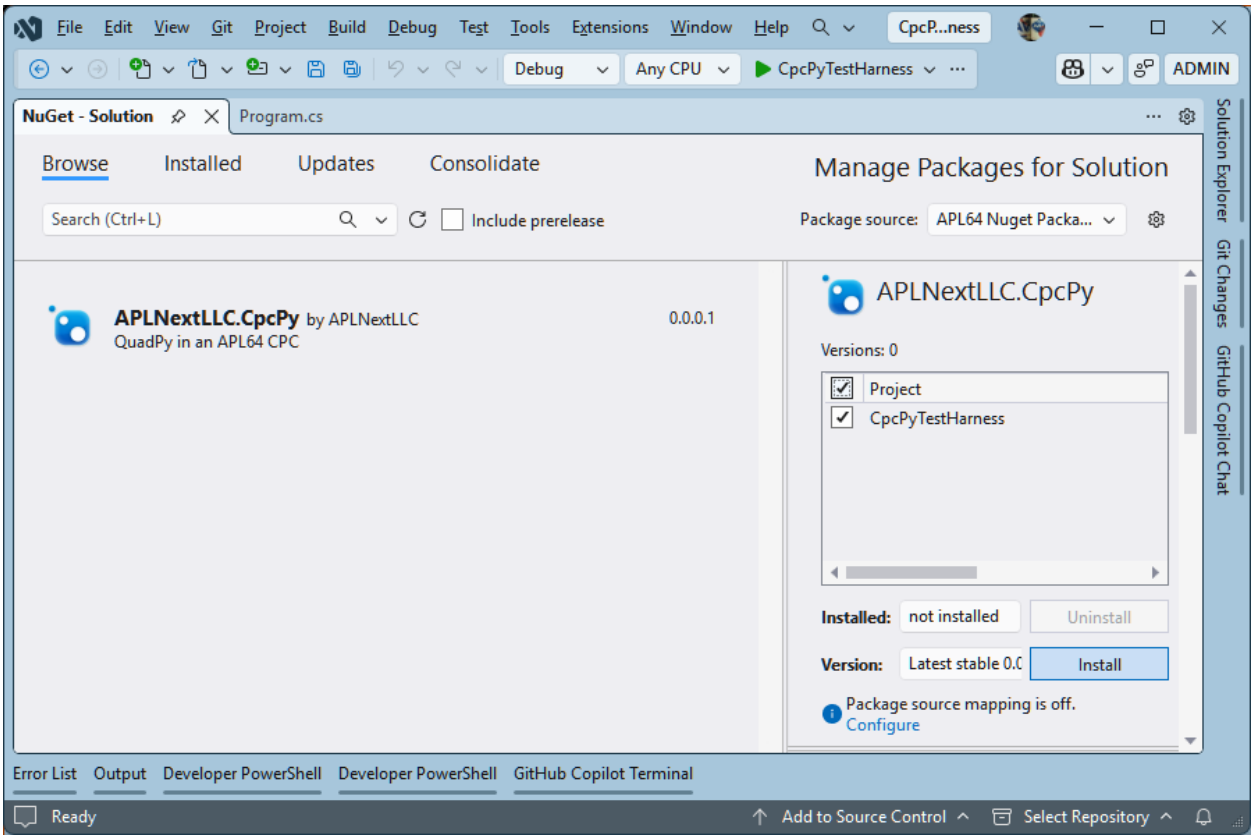
CPC Creation utility dialog:



.Net test harness for the APL64 CPC:

Create console project in Visual Studio 2026 targeting .Net 10.

Install the APL64 CpcPy Nuget package into the project:

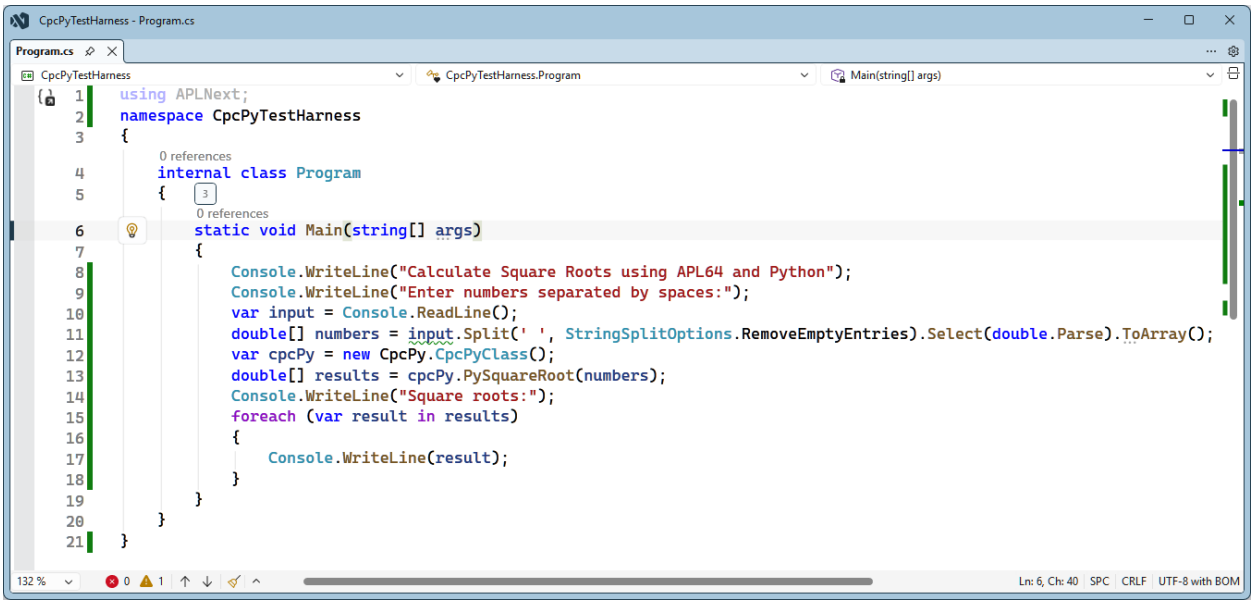


Edit the Program.cs fle to include the source code to:

- Get user input of a vector of doubles
- Create an instance of the CpcPy.CpcPyClass
- Use the APL64 public function, exposed as the .Net PySquareRoot() to calculate the square roots of the input vector
- Display the results

```
using APLNext;
namespace CpcPyTestHarness
{
  internal class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Calculate Square Roots using APL64 and Python");
      Console.WriteLine("Enter numbers separated by spaces:");
      var input = Console.ReadLine();
```

```
        double[] numbers = input.Split(' ',
StringSplitOptions.RemoveEmptyEntries).Select(double.Parse).ToArray();
        var cpcPy = new CpcPy.CpcPyClass();
        double[] results = cpcPy.PySquareRoot(numbers);
        Console.WriteLine("Square roots:");
        foreach (var result in results)
        {
            Console.WriteLine(result);
        }
    }
  }
}
```



Run (F5) the CpcPyTestHarness project:

□PY Development Challenges

Implementing effective data conversion between APL64 values and Python values is the major development challenge for the APL64 □PY system function. For example, Python supports certain data types in its base implementation. Python modules, such as numpy, have additional data types.

Feedback from □PY uses is essential information for the APL64 development team. Report you □PY experiences to support@apl2000.com to help improve APL64 for all users.