

ArcGIS ユーザーのための Python 基礎

ESRIジャパン株式会社



目次

- Python とは
 - Python とは
 - ArcPy とは
- 開発環境
 - 統合開発環境 (IDE)
 - Python shell とスクリプト ウィンドウ
 - 既存スクリプトの編集
 - IDLE からスクリプトの実行
- ArcGIS Python ウィンドウ の使用
 - ArcGIS Python ウィンドウ
 - ArcGIS Python ウィンドウの表示方法



目次

- Python 言語の基礎
 - コメント
 - モジュールのインポート
 - 変数
 - データ型
 - 文字列
 - 数値
 - リスト
 - タプル
 - 辞書
 - 関数



目次

- クラスとオブジェクト
- 構文
 - インデント
 - 条件分岐 (if else elif)
 - 繰り返し (while for)
 - 例外処理、try 構文
 - with 構文



Python とは？

Python とは

- プログラミング言語
 - タスクを自動化する
 - シンプルなコーディングで可読性の高いコードが記述できる
 - 世界中で親しまれているスクリプト言語
- 豊富なライブラリが提供されている
 - 数学・科学技術計算ライブラリが使用できる
 - ビッグデータ解析や機械学習にも使用されている
- オープンソース
 - ArcGIS Desktop では、9.x から導入



Language Rank	Types	Spectrum Ranking
1. Python	🌐 🖥️ 📱	100.0
2. C++	📱 🖥️ 📱	99.7
3. Java	🌐 📱 🖥️	97.5
4. C	📱 🖥️ 📱	96.7
5. C#	🌐 📱 🖥️	89.4
6. PHP	🌐	84.9
7. R	🖥️	82.9
8. JavaScript	🌐 📱	82.6
9. Go	🌐 🖥️	76.4
10. Assembly	📱	74.1

出展 : The Top Programming Languages 2018 (IEEE Spectrum)

Python とは

バージョンにおける留意点と ArcGIS

- 2 系と 3 系がある

- ArcGIS Desktop (Map) : 2系
- ArcGIS Pro : 3系



- 書き方に違いがある場合がある

- 3.x 系で Unicode に柔軟に対応できるようになった（日本語の扱いが容易になった）
- すべてにおいて互換性があるわけではないので書き方には注意が必要！
 - 例） print（2系ではステートメントだったが、3系では関数化された）

```
# Python 2系  
print "出力する文字列"
```

```
# Python 3系  
print("出力する文字列")
```

ArcPy とは

- Python スクリプトから地理的データの解析、変換、管理などを実行するための便利な関数およびクラスの集まり（パッケージ）
- ジオプロセッシング ツールを拡張するための標準ライブラリ
- ArcGIS 製品群 のさまざまなタスクを自動化





開発環境

開発環境

統合開発環境 (IDE)

- 統合開発環境 (IDE) は、Python コードの記述とテストを行うツールを組み合わせたソフトウェア アプリケーション
- IDE 内でコードを記述し、コードの構文とコード補完を支援する機能を利用可能
- アプリケーション内からコードを実行することができ、結果ウィンドウにフィードバックを提供することが可能



開発環境

統合開発環境 (IDE)

- 米国を含め Esri で利用されているもの

- 2.x 系 : PyScripter



- 3.x 系 : PyCharm, Visual Studio, VS Code



- IDLE

- Python をインストールすると標準で付属する統合開発環境



開発環境 (IDLE)

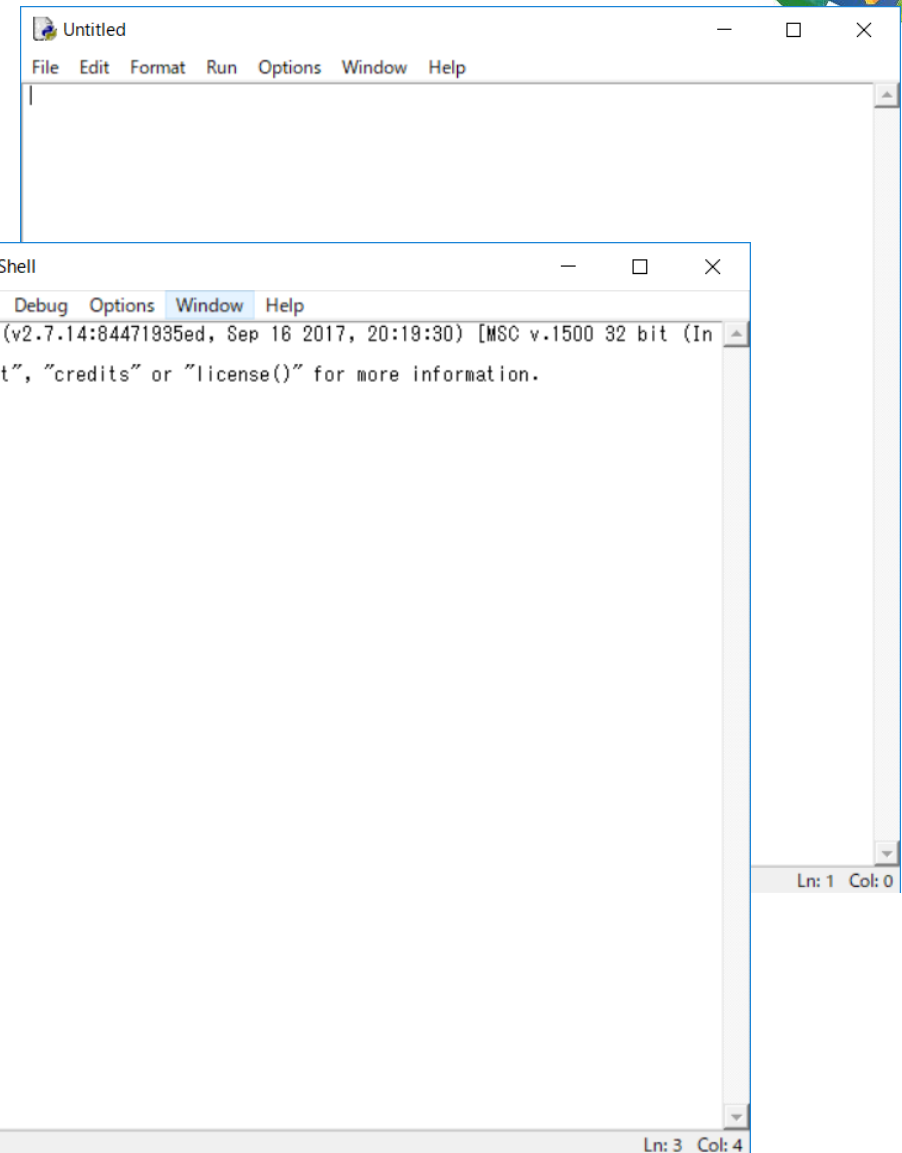
Python Shell と スクリプト ウィンドウ

- Python Shell は IDLE を起動したときの画面

- コードのテスト実行
- メッセージの表示

- スクリプト ウィンドウ

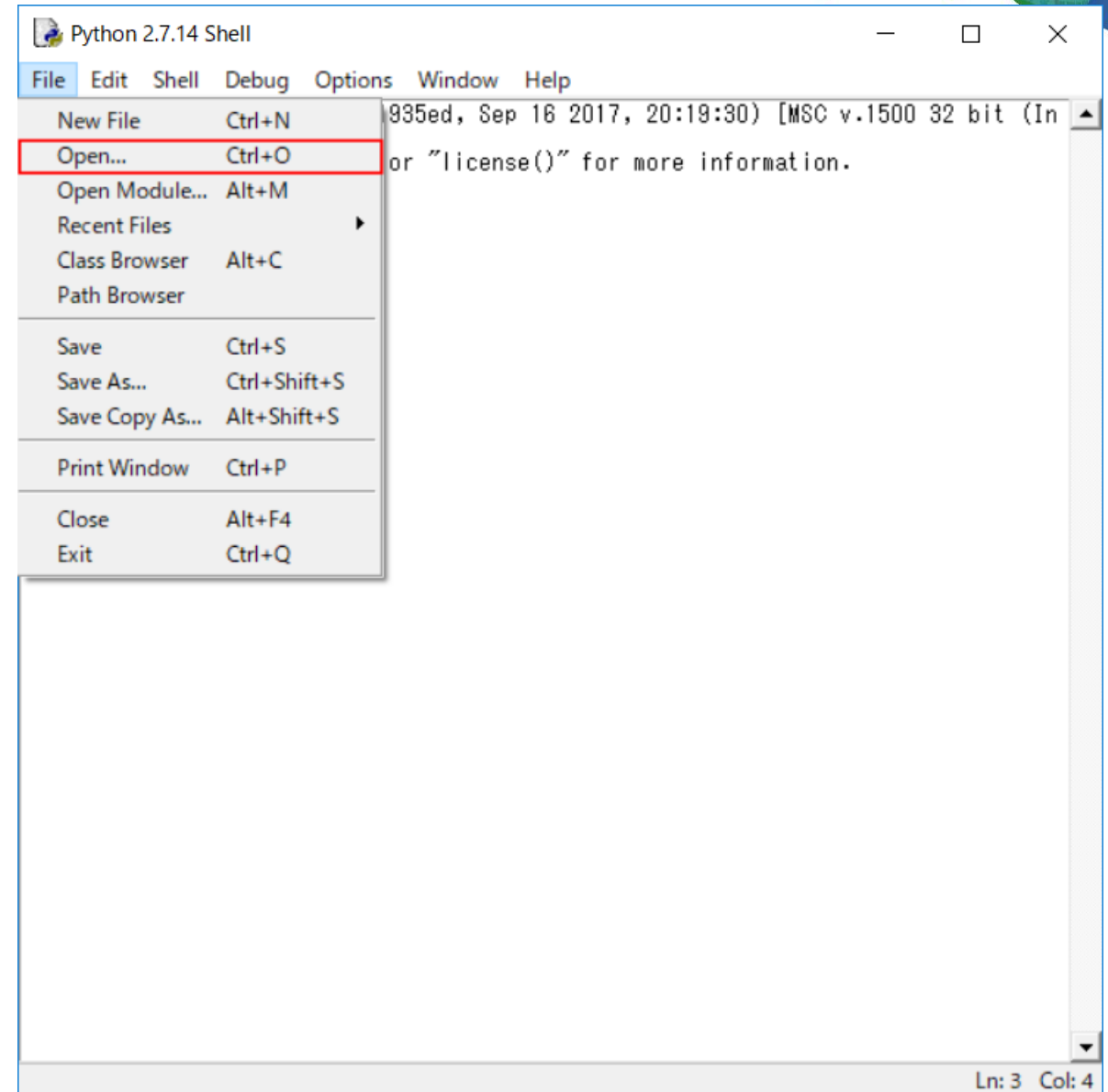
- コードの記述と保存
- [File] メニュー → [New File] で新規作成
- [File] メニュー → [Open] で既存のスクリプトを開く



開発環境 (IDLE)

既存スクリプトの編集

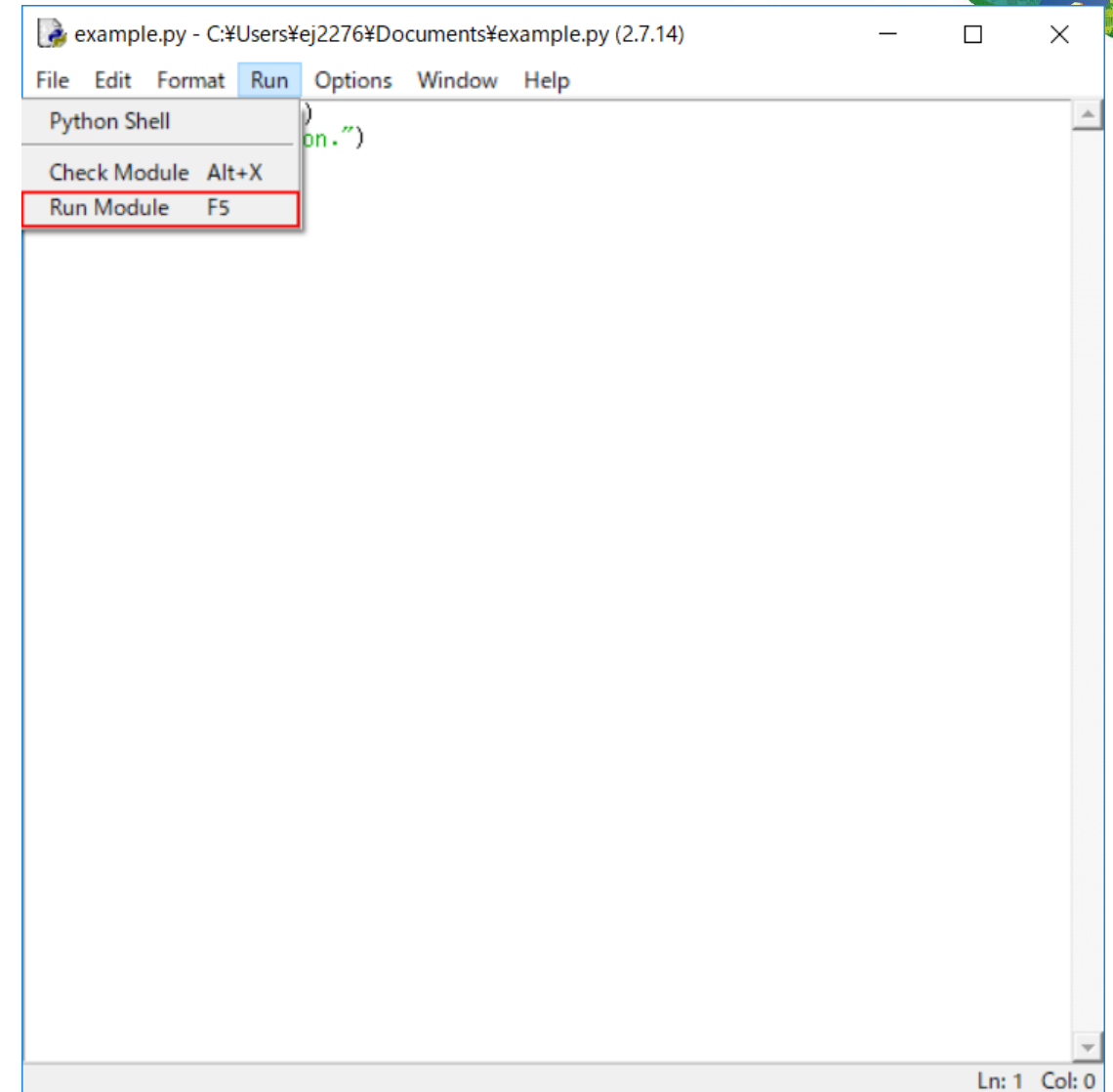
- IDLE を起動
- [File] メニュー → [Open] をクリック
- 既存のスクリプトを選択



開発環境 (IDLE)

IDLE からスクリプトの実行

- スクリプト ウィンドウの
[Run] メニュー → [Run Module]
をクリック
- Python Shell に実行結果が表示



ArcGIS Python ウィンドウ の使用

ArcGIS Python ウィンドウの使用

ArcGIS Python ウィンドウ




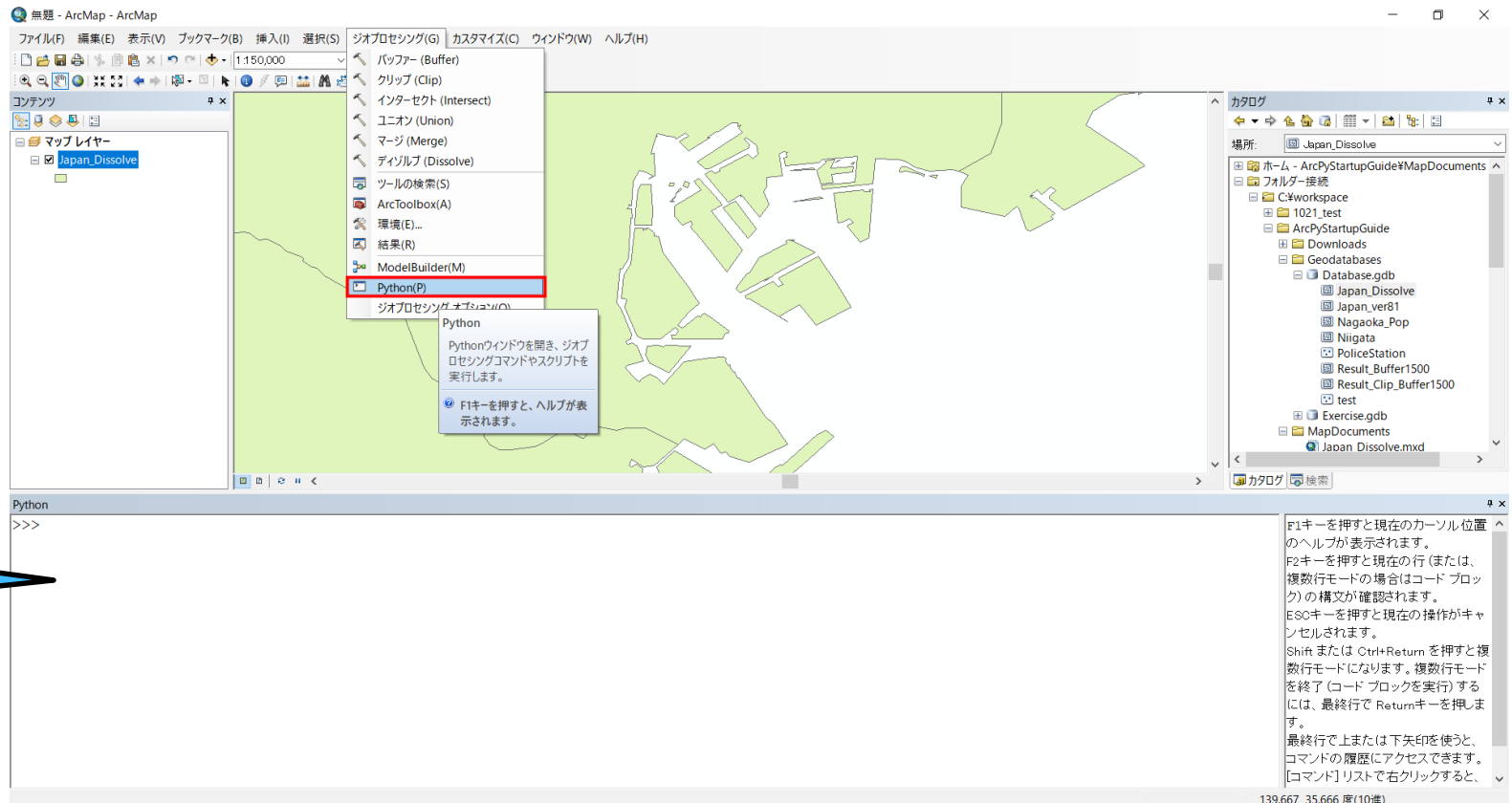
- ArcGIS 内で Python の機能が使用可能
- ArcMap や ArcGIS Pro からジオプロセッシング ツールを対話的に実行
- レイヤーやコンテンツ等を操作することが可能

ArcGIS Python ウィンドウの使用

ArcGIS Python ウィンドウの表示方法

- ArcMap 10.x

- [ジオプロセッシング] メニュー → [Python] をクリック
- または、ツールバーの Python ウィンドウ ボタン  をクリック



Python ウィンドウ

ArcGIS Python ウィンドウの使用

ArcGIS Python ウィンドウの表示方法

- Python ウィンドウで右クリックすると既存の Python ファイルの読み込みや Python ウィンドウで書いたコードの保存が可能

ここにコードを入力

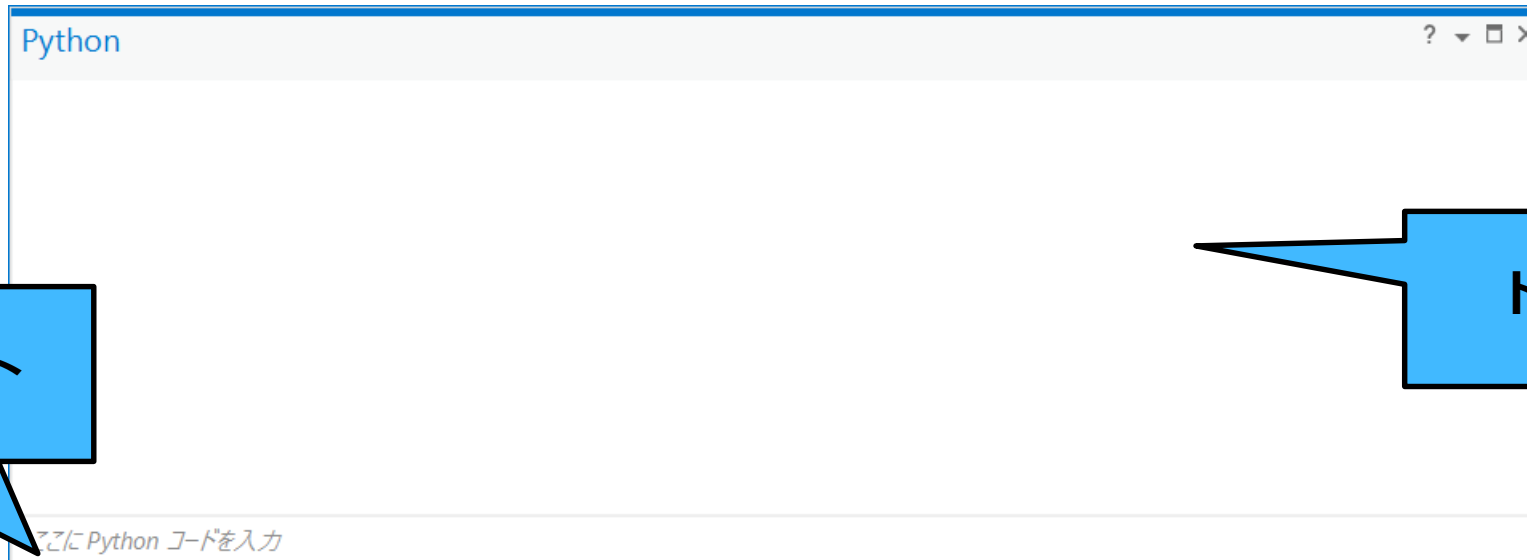


ヘルプと構文のウィンドウ

ArcGIS Python ウィンドウの使用

ArcGIS Python ウィンドウの表示方法

- ArcGIS Pro 2.x
 - [解析] タブ → [Python] をクリック



Python プロンプト

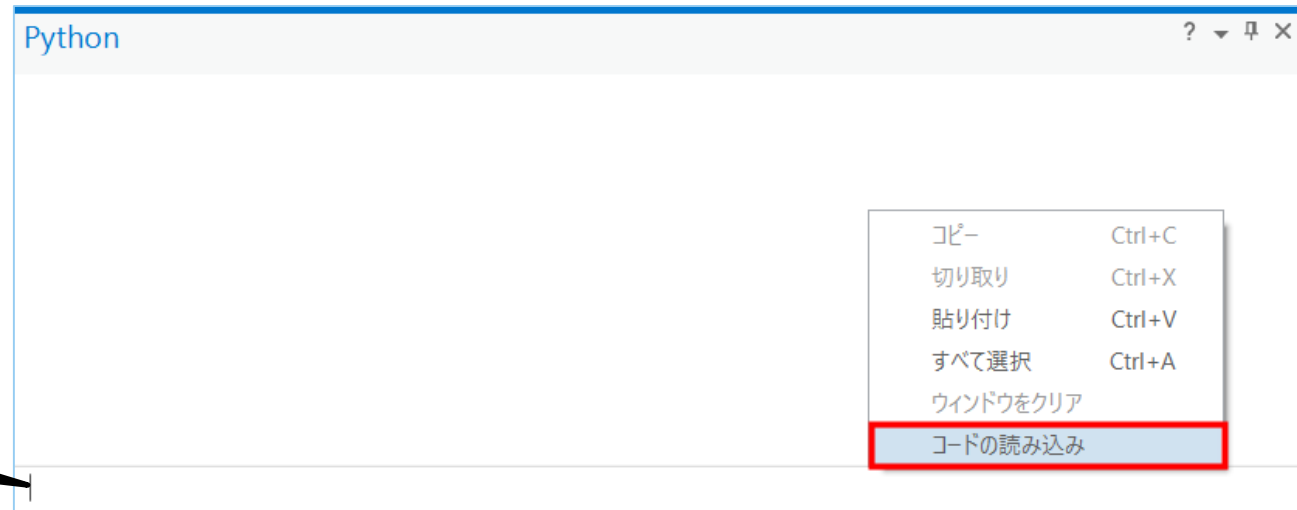
トランスクリプト

ArcGIS Pro Python ウィンドウ

ArcGIS Pro Python ウィンドウの使用方法

- Python ウィンドウは トランスクリプト と Python プロンプトで構成
 - コードは Python プロンプトに入力
 - ↓ Enter
 - コードが実行され、トランスクリプトセクションに移動
 - メッセージやエラーはトランスクリプトセクションに表示される
- Python プロンプトで右クリック → [コードの読み込み] をクリック
既存の Python ファイルのコードを Python ウィンドウに読み込むことが可能

ここにコードを入力

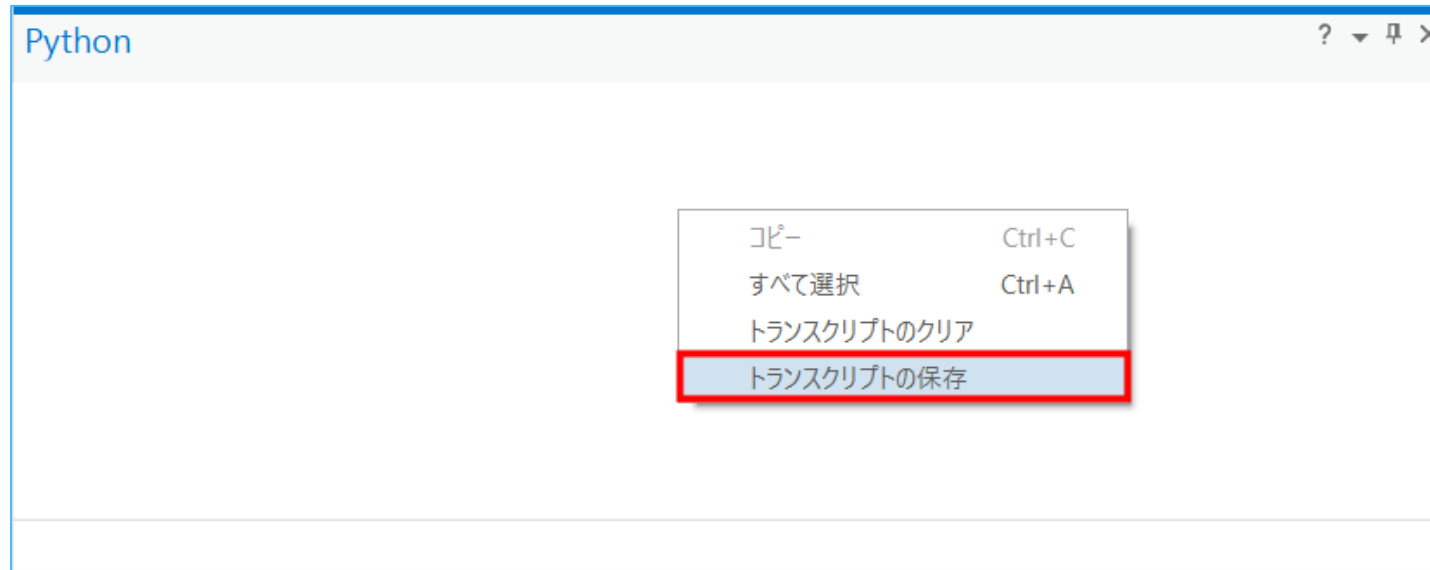


ArcGIS Pro Python ウィンドウ

ArcGIS Pro Python ウィンドウの使用方法



- トランスクリプトで右クリック → [トランスクリプトの保存] をクリック
Python ウィンドウで書いたコードを保存できる



- PyCharm など他の統合開発環境 (IDE) で作業する際に便利

Python 言語の基礎

Python 言語の基礎

コメント

- プログラムに関する注釈などを記述できる
- 行単位のコメントは # をつけて記述
- # 以降、行末まで実行結果に影響なし

```
# Python という文字を出力  
print("Python")
```

- 複数行にまたがるコメントは
''' または """ で囲まれた部分がコメントアウトされる

```
''''  
複数行のコメント文が  
コメントアウトされます。  
''''
```



Python 言語の基礎

モジュールのインポート

● モジュール

- コードが記述され、機能が格納されているファイル (*.py)
- Python をインストールすることでさまざまなモジュールも利用することができる (組み込みモジュール)
- インポートすることで利用可能
- 多くの関数はモジュールからインポートが必要



Python 言語の基礎

モジュールのインポート

● モジュールの例

● math モジュール : 数学関数

```
import math
print(math.sqrt(64))
8.0

print(math.pow(10, 2))
100
```

math.sqrt(x) ... x の平方根を計算

math.pow(x,y) ... x の y 乗を計算

● string モジュール : 文字列を操作する関数

```
import string
print(string.split("a,b,c,d"))
['a','b','c','d']

print(string.upper("abcde"))
ABCDE
```

string.split ... 文字を区切りリストを作成

string.upper ... 大文字を返す



Python 言語の基礎

モジュールのインポート



● モジュールの例

- os.path モジュール : パス名の操作
 - os.path.join ... パスやファイル名などを結合
 - os.path.split ... ファイル名を境にパスを分割

```
import os.path

path = "¥A¥B¥C¥example.py"

print(os.path.split(path))
('¥A¥B¥C', 'example.py')
```

```
import os.path

path = "¥A¥B¥C"

file = "example.py"
print(os.path.join(path, file))
('¥A¥B¥C¥example.py')
```

Python 言語の基礎

モジュールのインポート

- **import**
 - モジュールをインポートする
- **from..import**
 - モジュール内の特定の関数をインポートする
- **import..as**
 - モジュールに任意の名前をつけてインポートする

```
import math
import string
import os
```

```
from math import sqrt
print(sqrt(64))
```

```
import math as m
print(m.sqrt(64))
```



Python 言語の基礎

変数

- データを格納する入れ物

- 変数名 = 値 で作成
- 文字列や数値の情報を格納し、スクリプトの指示に使用
- 変数名には日本語や予約語は使えない
- 大文字と小文字が区別される

```
A = 1  
a = 2
```

←それぞれ異なる変数

- 変数には様々なデータ型を格納できる

変数名としては使えない

予約語

and	if
as	import
break	in
class	is
continue	not
def	or
del	pass
elif	print
else	raise
except	return
finally	try
for	while
from	with
	など

データ型

文字列

- 文字列 (str)

- 文字で構成されたテキストデータ
- シングル クォーテーションやダブル クォーテーションで囲んだもの

```
a = 'python'
```

```
a = "python"
```

- トリプル クォーテーションは改行を含む複数行の文字列を記述できる

```
a = '''  
python  
example'''
```

```
a = """  
python  
example"""
```

- 文字列の連結

- 「+」 演算子を用いて 2 つの文字列を連結できる

```
a = "abcd"  
b = "efgh"  
a + b  
abcdefgh
```

データ型

文字列

- マルチバイト文字の記述を行うために文字コードを指定

- スクリプトの 2 行目までにコメントで指定

```
#coding:cp932
```

```
#coding:utf-8
```

- 指定する文字コードはファイルに依存

- Unicode 文字列

- Python で日本語を扱う際に使用
- Python 3 系では Unicode へ統一
- Python 2 系では文字列の前に u と記述

```
print u"日本語文字列"
```



データ型

数値

- 数値
 - 数を表現するデータ型
 - int
 - 整数を表す
 - float
 - 浮動小数点数（小数）を表す
 - complex
 - 複素数を表す



データ型

数値



- 計算

- 四則演算 (+、-、*、/)

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> 17 / 3      # 除算は常に浮動小数点数 (float) を返す
5.666666666666667
```

- 剰余 (%), 小数切り捨て (//)

```
>>> 17 // 3      # // 演算子は小数を切り捨てた整数 (int) を返す
5
>>> 17 % 3      # % は余りを返す
2
```


データ型

数値

- 計算

- 冪乗 (**)

```
>>> 5 ** 2 # 5 の 2 乗  
25  
>>> 2 ** 7 # 2 の 7 乗  
128
```

- 変数に代入して計算

```
>>> width = 10  
>>> length = 20  
>>> width * length  
200
```



データ型 リスト

● リスト

- オブジェクトの集合で、複数の値を 1 つの変数で管理
- [] 内の要素をカンマで区切って記述

```
a = [0,1,2,3]  
a  
[0, 1, 2, 3]
```

- 要素の変更が可能



データ型 リスト

リストの要素にアクセスする方法

- リストは前から順にインデックス (0,1,2...) で管理されている
- 特定の値を取り出すことができる

```
>>> a = [0,1,2,3]
>>> a[0]
0
```

a の 0 番目のデータ

```
>>> a = [0,1,2,3]
>>> a[-1]
3
```

a の後ろから 1 番目のデータ

```
>>> a = [0,1,2,3]
>>> a[3]
3
```

a の 3 番目のデータ

```
>>> a = [0,1,2,3]
>>> a[-2]
2
```

a の後ろから 2 番目のデータ



データ型 リスト

- リストの連結

```
>>> a = [0,1,2,3]
>>> a + [4,5,6,7]
[0, 1, 2, 3, 4, 5, 6, 7]
```

- リストの要素の変更

```
>>> b = [1,4,9,18,25]
>>> b[3] = 16
>>> b
[1, 4, 9, 16, 25]
```

b の3 番目のデータ (18) を16に変更

- その他の操作の例

関数・メソッド	機能
len	リストの要素数を返す
del リスト[i]	i 番目の要素を削除
append(x)	リストの末尾に値 x を追加
index(x)	要素 x を探しその位置を返す
insert(i, x)	i 番目に 値 x を挿入
pop	リストの末尾の値を削除

データ型 タプル

● タプル

- オブジェクトの集合
- 複数の値を 1 つの変数で管理
- () 内の要素をカンマで区切って記述

```
>>> a = (0,1,2,3)
>>> a
(0, 1, 2, 3)
```

- 要素を変更することはできない
- リスト同様 () 内の要素はインデックス (0,1,2,3...) で管理
- 特定の値の取り出すことが可能

```
>>> b = (1,4,9,18,25)
>>> b[3] = 16
>>> b
```

```
>>> b = (1,4,9,18,25)
>>> b[3] = 16
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    b[3] = 16
TypeError: 'tuple' object does not support item assignment
```

データ型 辞書

● 辞書

- リスト、タプル同様、複数の値を 1 つの変数で管理
- 複数のキーと値をペアでひもづけ
- キーとそれに対応する値を {} 内にペアで格納

```
x = { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4 }
```

- キーを検索すると値が得られる
- データの参照

```
x = { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4 }  
x['a']  
1
```

← 辞書型データの作成
← ['キー'] で参照



関数



- 関数

- 特定の処理をまとめたもの
- Python に用意されている関数 (組み込み関数) を利用可能

- 引数

- 関数に渡す値

- 戻り値

- 引数を使って関数が処理を行った後に結果として返ってくる値
- return を使用

関数

関数の定義

- 関数を自分で作ることができる
- `def` を使用して関数を定義
- `def 関数名(引数) :` で記述

#引数のない関数

```
def hello():  
    print("Hello")
```

```
hello()  
Hello
```

#引数のある関数

```
def test(a,b):  
    print(a*b)
```

```
test(2,3)  
6
```



関数

関数の定義

- return の使用で戻り値を返す
- return を使わないと None と返される

```
def test(a,b):  
    c = a * b  
  
print(test(3,5))  
  
None
```



```
def test(a,b):  
    c = a * b  
    return c  
  
print(test(3,5))  
  
15
```



関数

関数の定義

- **return** は他の関数で戻り値を使う場合に用いる

```
def Hello():                # 関数を定義
    return "Hello"          # 戻り値

def Python():               # 戻り値を利用した関数を定義
    a = Hello()
    return a + " " + "Python"

print(Python())

Hello Python
```

クラスとオブジェクト

クラスとオブジェクト

オブジェクト



- データや関数等をひとつにまとめたもの
- Python のデータはすべてオブジェクトに含まれる
 - 変数、データ型、モジュール、関数、メソッドなどはすべてクラスから作られたオブジェクト
- オブジェクトのデータ型は `type` 関数で調べることができる

```
type("python")  
<type 'str'>
```

```
type(1)  
<type 'int'>
```

クラスとオブジェクト

クラス

- オブジェクトの定義を表すもの
- クラスからオブジェクトを作成して利用
- インスタンス…クラスから作ったオブジェクト
- よく使う処理を `class` で作成しておくと同じコードを何回も書く必要がない
- `class` 文で定義

```
class クラス名 :  
    クラスの内容
```

```
class クラス名:  
    def __init__(self, 変数名1、変数名2):  
        self.変数名1 = 変数名1  
        self.変数名2 = 変数名2
```





構文

構文

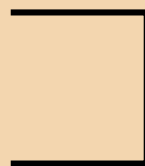
- プログラミングに指示を行う文
- 式との組み合わせでスクリプトを構成
- 単純文
 - import、print (2系のみ)...
- 複合文
 - 条件分岐 (if..elif..else)
 - 繰り返し、ループ (for..in、while)



構文 インデント

- Python ではインデントを用いてコードブロックを定義
- 同じ数の空白でインデントされた文が同じブロックとみなされる

```
n = 3  
if n == 5:  
    print ("aaaaa")  
    print ("bbbbbb")
```



if 文の対象

```
print ("ccccc")
```



if 文の対象ではない

構文

条件分岐 (if else elif)

- 条件によって実行する処理を分ける
- 指定した条件が成立した場合のみ処理の実行

- if 構文

if [条件式 1]:

[条件式 1 が True のときに行う処理]

elif [条件式 2]:

[条件式 1 が False で条件式 2が True のときに行う処理]

else:

[if 文の条件式 1 も elif の条件式 2 もどちらも False のときに行う処理]

条件の末尾には
「:」コロンを記述

条件を満たす処理、満たさない処理は
インデントを下げた状態で記述



構文

条件分岐 (if else elif)

- if 構文

- if
 - 条件を満たすとき
- else
 - 条件式を満たさない場合の処理

```
a = 49
if a >= 50:                # a が 50 以上であるとき
    print ("a は 50 以上です。")
else:                      # それ以外のとき
    print ("a は 50 未満です。")
```



構文

繰り返し処理 (ループ)

- while 構文
 - 条件付き繰り返し
 - 条件式を満たしている (真である) 間処理を繰り返す

条件の末尾には
「:」コロンを記述

while [条件式]:
繰り返し行う処理

繰り返し行う処理は
インデントを下げた状態で記述

```
n = 1
while n <= 5:  # n が5以下でのとき繰り返し処理
    print(n)    # n を出力
    n += 1      # n に 1 を足す

1
2
3
4
5
```



構文

繰り返し処理 (ループ)

- for 構文
 - 範囲付き繰り返し
 - リストの要素を順に取り出すことなどが可能

変数の後ろに「in」を記述

オブジェクトの後ろに「:」コロンを記述

for 変数 in オブジェクト :
繰り返し行う処理

繰り返し行う処理は
インデントを下げてもろえて記述

```
a = [0,1,2,3,4] # リストの作成
for i in a:      # リストの要素を繰り返し出力
    print(i)

0
1
2
3
4
```

構文 エラーと例外

- 構文エラー

- SyntaxError と表示されるエラー
- 文法的な間違いによって発生するエラー
- プログラムの書き方が間違っている

- 例外

- プログラム実行時に発生するエラー
- 文法的に正しくてもエラーが発生することがある



構文

例外処理

- 一般的な例外

- **NameError**

- 未定義の変数などを参照した場合に発生

- **IndexError**

- リスト内の要素をインデックスを使用して参照した際に、要素数を超える数を指定した場合に発生

- **TypeError**

- 関数の引数として、正しくない型を設定した場合に発生
- 文字列と数値を連結させるなど、演算子を使った演算をする場合に発生

- **ImportError**

- 存在しないモジュールをインポートしようとした場合に発生

構文

例外処理



- **try – except**
 - 例外が発生したときにどのような処理を行うかを記述

```
try:  
    実行したい処理  
except エラー名:  
    例外発生時に行う処理
```

- **try** の中で処理を実行
例外が発生しなければ **except** はスキップ
発生した場合は **except** 文が実行される

構文

例外処理

- **else**

- 例外が発生しなかった場合に行いたい処理

- **finally**

- 例外の発生に関わらず必ず処理を行いたい場合に使用

```
try:  
    実行したい処理  
except エラー名:  
    例外発生時に行う処理  
else :  
    例外が発生しなかったときに行う処理  
finally:  
    例外の有無に関わらず必ず実行したい処理
```



構文

with 構文

- with 構文

- 前処理、後処理を必要とするオブジェクトを扱う際に簡易的に記述が可能
- 開始時、終了時の処理を必ず実行
- ファイルを扱う時に使用
 - ファイルを開く
 - ファイルを閉じる



構文

with 構文

- with を使わない場合、ファイルを閉じる close 処理が必要

```
# with を使わない場合
```

```
f = open("example.txt", "r")  
print(f.read())  
f.close()
```

- with を使うと close 処理を省略できる

```
# with を使う場合
```

```
with open("example.txt", "r") as f:  
    print(f.read())
```

