Python Tutorial – Part I: Introduction

Mark A. Austin

University of Maryland

austin@umd.edu

ENCE 201, Fall Semester 2023

February 7, 2024

Overview

Introduction

What is Python?

What is Python?

The Origins of Python

The Python programming language was initially written by Guido van Rossum in the late 1980s and first released in the early '90s. Its design borrows features from C, C++, Smalltalk, etc.

The name Python comes from Monty Python's Flying Circus.

Version 0.9 was released in February 1991. Fast forward to 2022, and we are up to Version 3.11.

What is Python?

Features: Advertising ...

- Designed for quick-and-dirty scripts, reusable modules, very large systems.
- Object-oriented. Very well-designed. Well documented.
- Large library of standard modules and third-party modules.
- Works on Unix, Mac OS X and Windows.
- Python is both a compiled and interpreted language. Python source code is compiled into a bytecode format.
- Integration with external C and Java code (Jython).

What is Python?

Strengths of Python: (**easy to get started**)

- Provides an approximate superset of MATLAB functionality.
- Modern language with good support for object-oriented program development.
- But, Python doesn't force users to think in term of objects from the very beginning ...
- Open source. Licenses are free.

Weaknesses of Python: (**throw away code**)

- Behind the scenes, everything is an object. The language design is not as clean (logical) as Java.
- Python provides users with considerable freedom to mix-and-match data types. Code might not scale well, and could become very difficult to debug/maintain.
- Language versions are not backwards compatible. Ugh !!!!

What is Python?

Third-Party Tools:

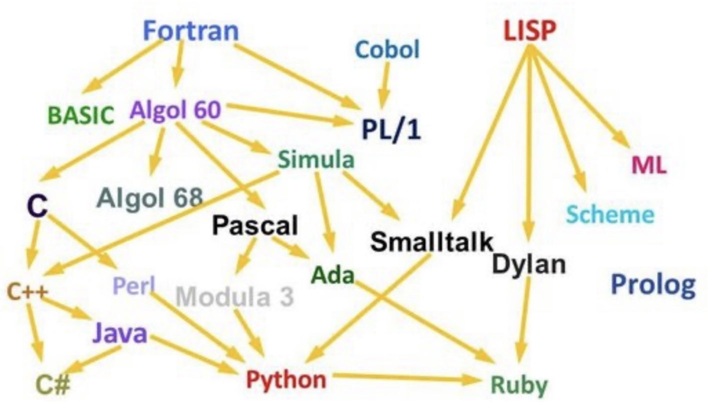
- pip3 is a command-line tool for installing Python modules.
- csv reads/writes comma-separated data files.

Many Third-Party Modules:

- NumPy is a language extension that defines the numerical array and matrix type and basic operations on them.
- SciPy uses numpy to do advanced math, signal processing, optimization, statistics, etc.
- Matplotlib provides easy-to-use plotting Matlab-style.
- Tensorflow is an open source machine learning platform developed by Google.

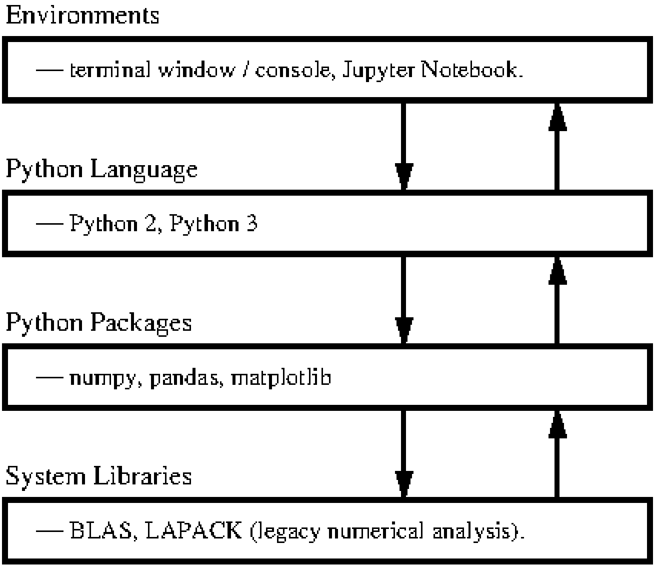
What is Python?

Graph of Feature Dependencies Among Computer Languages



Python Language: Borrows from C++, Java, Smalltalk, ..

Framework for Scientific Computing



Program Development with Python

First Steps: Working with the Terminal

Terminal Window (Console)

The standard approach runs a program directly through the Python intepreter.

Terminal — Python — 112×26

```
/Users/austin 872>> python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = [ 1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3, 4, 5, 6]
>>> print(type(a))
<class 'list'>
>>> b = [ (1, 2), (3, 4), (5, 6) ]
>>> print(b)
[(1, 2), (3, 4), (5, 6)]
>>> print( type(b) )
<class 'list'>
>>> import numpy as np
>>> c = np.array(b)
>>> print(c)
[[1 2]
 [3 4]
 [5 6]]
>>> print( type(c) )
<class 'numpy.ndarray'>
>>>
```

First Steps: Using Python as a Calculator

You can type expressions in the command window, e.g.,

```
>>> 2 + 3/4*5
5.75
>>>
```

Expressions are evaluated according to predefined priorities:

- Evaluate quantities in brackets,
- Evaluate powers $2 + 3^2 \rightarrow 2 + 9 \rightarrow 11$.
- Evaluate * /, working left to right (i.e., 3*4/5 $\rightarrow$ 12/5),
- Evaluate + -, working left to right (3+4-5 $\rightarrow$ 7-5),

Program Development

Step-by-Step Procedure:

1. Write, compile, fix, run, fix, run, validate → success!
2. Interpreted and compiled languages.

Program Control Structures:

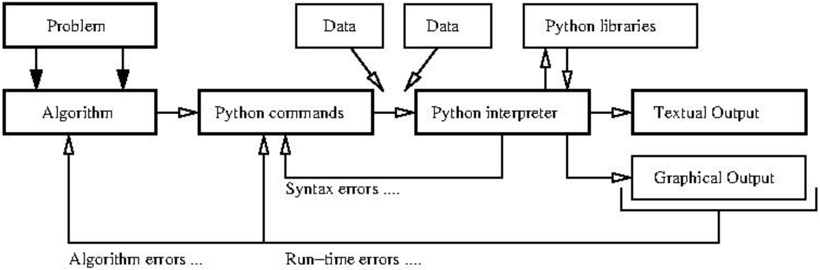
1. Logical and relational expressions
2. Selection constructs
3. Looping constructs

Program Input and Output:

1. Reading variables from the keyboard and files.
2. Formatted output of variables

First Steps: Working with the Terminal

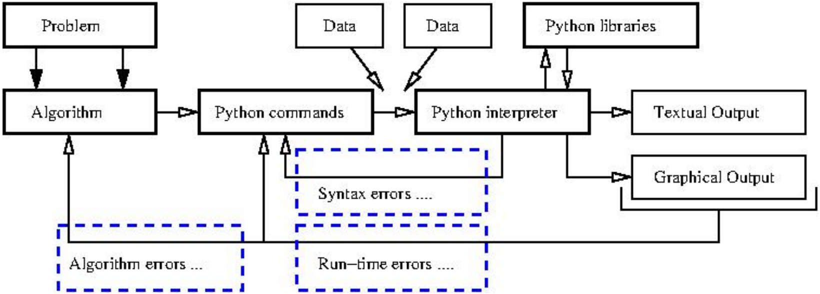
Program Development in the Terminal Window:



Step-by-Step Procedure:

1. Write, compile, fix, run, fix, run, validate → success!

First Steps: Fixing Mistakes



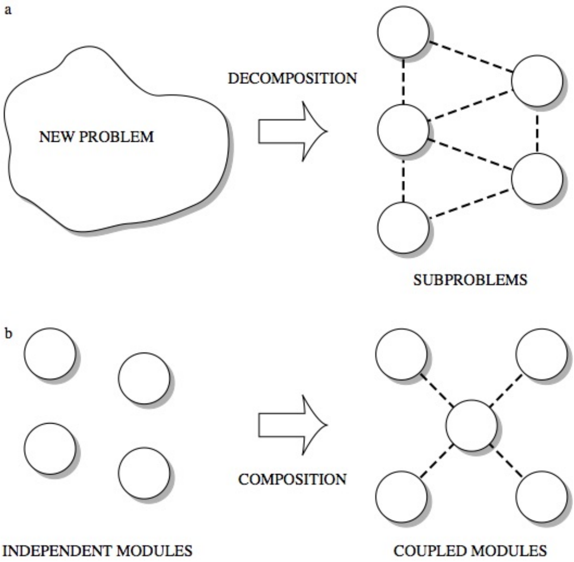
1. Syntax Errors: Check your typing ...
2. Runtime Errors: Program runs, but you have divide by zero and/or NaNs, etc.
3. Algorithm Errors: Does your program solve the right problem?

First Steps: Program Evaluation

Program Evaluation

- Robustness (does it work?)
- Accuracy and Efficiency (speed).
- Ease of Implementation (cost).

Top-Down and Bottom-Up Program Design



Top-Down and Bottom-Up Program Design

Advantages/Disadvantages of Top-Down Development

- Can customize a design to provide what is needed and no more.
- Start from scratch implies slow time-to-market.

Advantages/Disadvantages of Bottom-up Development

- Reuse of components enables fast time-to-market.
- Reuse of components improves quality because components will have already been tested.
- Design may contain (many) features that are not needed.

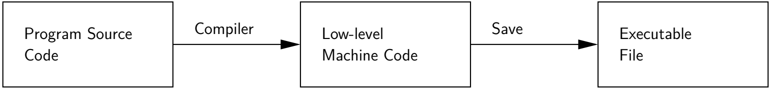
Interpreted and Compiled Programming Languages

Interpreted Programming Languages:

- High-level statements are read one by one, and translated and executed on the fly (i.e., as the program is running).

Compiled Programming Languages:

- A compiler translates the computer program source code into lower level (e.g., machine code) instructions.



- High-level programming constructs (e.g., evaluation of logical expressions, loops, and functions) are translated into equivalent low-level constructs that a machine can work with.

Interpreted and Compiled Programming Languages

Benefits of Compiled Code:

- Compiled programs generally run faster than interpreted ones.
- This is because an interpreter must analyze each statement in the program each time it is executed and then perform the desired action.

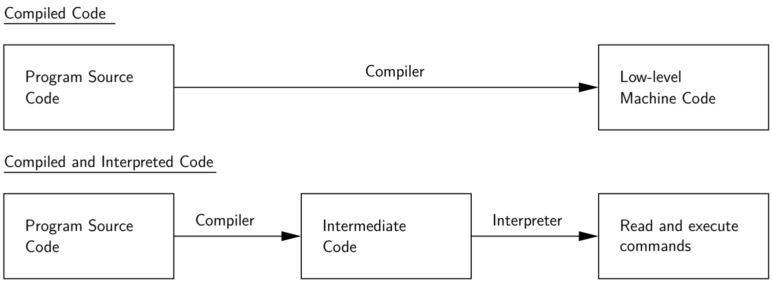
Benefits of Interpreted Code:

- Interpreted programs can modify themselves by adding or changing functions at runtime.
- Cycles of application development are usually faster than with compiled code because you don't have to recompile your application each time you want to test a small section.

Interpreted and Compiled Programming Languages

Modern Interpreter Systems

Transform source code into a lower-level intermediate format.
Interpreter then executes commands.



Examples: Java and Python (even MATLAB).

First Steps: So What's Next?

Things to Learn:

- Should I use an Integrated Development Environment?
- How are numbers stored inside the computer?
- How do variables work?
- How do vectors and matrices work?
- How do list, dictionaries and sets work?
- What's in the Python Programming Language?
- How to apply Python to solution of numerical problems?
- Where can I go for help?

Integrated

Development Environments

(Simplifying Program Development)

Integrated Development Environments

Integrated Development Environments

An Integrated Development Environment (IDE) is a software application that provides comprehensive support to computer programmers for software development.

State-of-the-art IDEs provide tools for:

- Syntax highlighting, editing source code, automation of program build, and code debugger.
- Program compilation (interpretation) and execution (run).

Two IDE's for Python:

- Visual Studio Code (for program development).
- Jupyter Notebook (web-based authoring of python documents).

Visual Studio Code

Visual Studio Code (vscode)

Visual Studio Code (vscode) is a source code editor for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion and code refactoring.

Standard Use Cases:

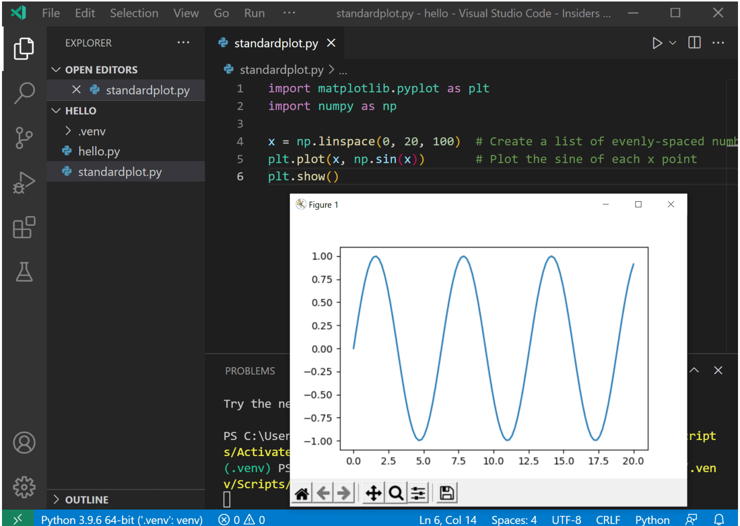
- Edit, debug, run, debug, run, test.
- Develop desktop apps.
- Numerical and scientific computing.

Advanced Use Cases:

- Deploy code to the cloud (Github).

Visual Studio Code

Graphical Interface



Jupyter Notebook

Jupyter Notebook (Web-based Application)

Web-based authoring of documents that combine live code with narrative text, equations and visualization.

To install Jupyter Notebook:

```
prompt >> pip3 install jupyter
```

To run Jupyter Notebook:

```
prompt >> jupyter notebook
```

Jupyter Notebook

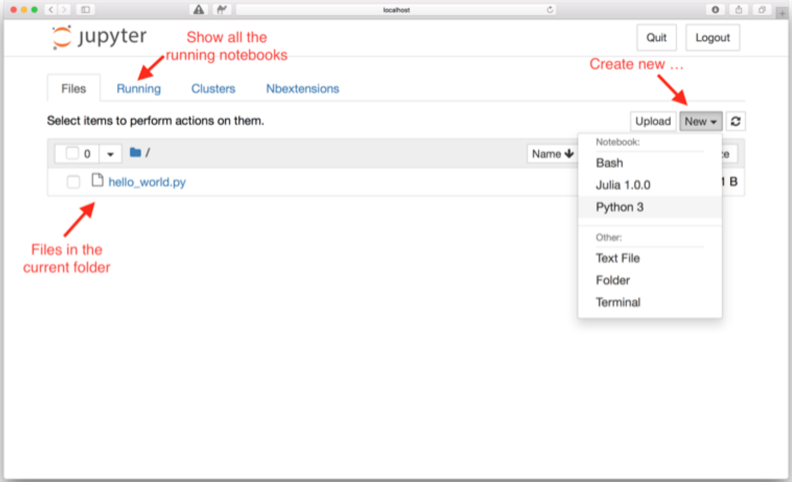
Use Cases:

- Data cleaning and transformation.
- Numerical simulation.
- Statistical modeling.
- Data visualization.
- Machine learning.

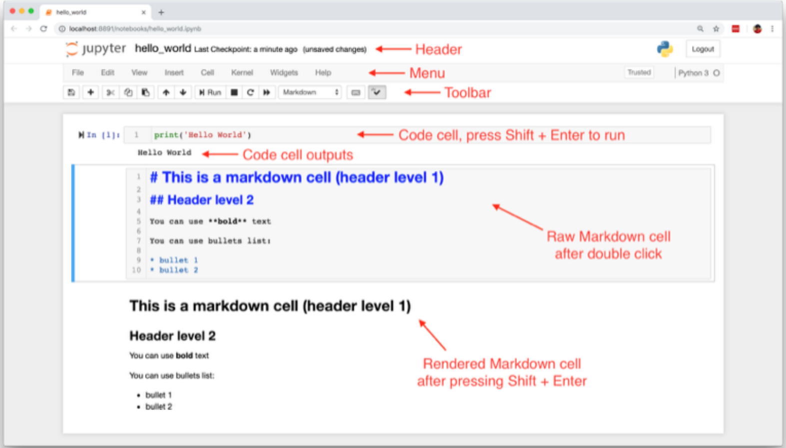
Jupyter Notebook File Format:

- File format is JSON-based with extension .ipynb (named after projects predecessor IPython).
- Supports documents containing text, source code, rich media data and metadata.

Jupyter Notebook User Interface



Jupyter Notebook User Interface



Jupyter Notebook Cells and Code Execution

Jupyter Notebook Cells:

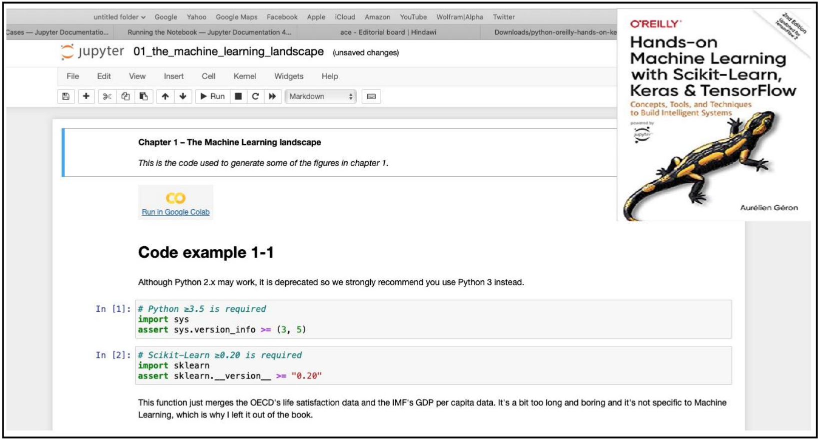
- **Code Cells:** Allows for development and editing of new code, with syntax highlighting and tab completion.
- **Markdown Cells:** Document the computational process with the Markdown language (a simple way to perform text markup). Can also include mathematics with LaTeX notion.
- **Raw Cells:** Provide a place in which you can write output directly.

Code Execution:

- When a code cell is executed, the code is sent to the kernel associated with the code.
- Results are returned to the computation and then displayed.

Jupyter Notebook and Machine Learning

Jupyter Notebook (Machine Learning with TensorFlow)

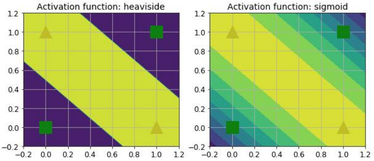


Jupyter Notebook and Machine Learning

Jupyter Notebook (Machine Learning with TensorFlow)

In [7]:
```python
def heaviside(z):
    return (z >= 0).astype(z.dtype)

def mlp_xor(x1, x2, activation=heaviside):
    return activation(-activation(x1 + x2 - 1.5) + activation(x1 + x2 - 0.5) - 0.5)
```

In [8]:
```python
x1s = np.linspace(-0.2, 1.2, 100)
x2s = np.linspace(-0.2, 1.2, 100)
x1, x2 = np.meshgrid(x1s, x2s)

z1 = mlp_xor(x1, x2, activation=heaviside)
z2 = mlp_xor(x1, x2, activation=sigmoid)

plt.figure(figsize=(10,4))

plt.subplot(121)
plt.contourf(x1, x2, z1)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: heaviside", fontsize=14)
plt.grid(True)

plt.subplot(122)
plt.contourf(x1, x2, z2)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: sigmoid", fontsize=14)
plt.grid(True)
```



Data Types

(Data Types in Python)

Integers

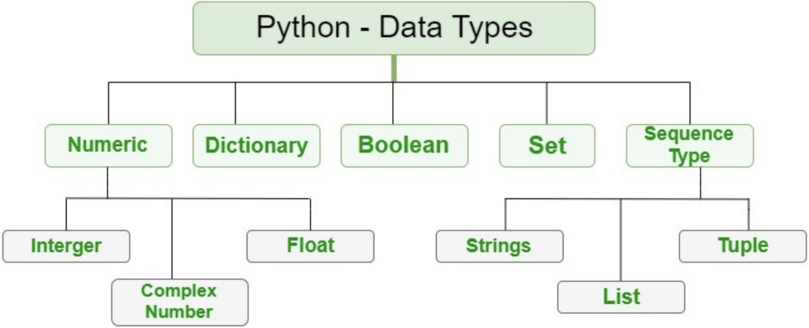
Requirements for storing 4 types of integer:

Type	Contains	Value	Size	Range and Precision
byte	Signed integer	0	8 bits	-128/127
short	Signed integer	0	16 bits	-32768/32767
int	Signed integer	0	32 bits	-2147483648/2147483647
long	Signed integer	0	64 bits	-9223372036854775808 / 9223372036854775807

Note. A 32 bit integer has $2^{32} \approx 4.3$ billion permutatons $\rightarrow$ a working range $[-2.147, 2.147]$ billion.

Builtin Data Types

Everything in Python is an object – there is no notion of primitive datatypes, e.g., as found in Java.



Builtin Data Types

```
dtype             Description
================================================================
Text Type:        str
Numeric Types:    int, float, complex
Sequence Types:   list, tuple, range
Mapping Type:     dict
Set Types:        set, frozenset
Boolean Type:     bool
Binary Types:     bytes, bytearray, memoryview
None Type:        NoneType
================================================================
```

Example 1: Getting an int data type ...

```
a = 1
print ( type(a) )
```

Output:

```
< class 'int' >
```

Builtin Data Types

Example 2: Float, complex, boolean, string and list types ...

```
b = 1.5                  # <-- define float ...
print ( type(b) )
c = 1.0 + 1.5j           # <-- define complex ...
print ( type(c) )
d = True                 # <-- define boolean ...
print ( type(d) )
e = "this is a string"   # <-- define string ...
print ( type(e) )
f = ["A, "B", "C", "D"]  # <-- define list ...
print ( type(f) )
```

Output:

```
< class 'float' >
< class 'complex' >
< class 'bool' >
< class 'str' >
< class 'list' >
```

Builtin Data Types

Example 3: Size of basic data types ...

```
print ( sys.getsizeof(a) )
print ( sys.getsizeof(b) )
print ( sys.getsizeof(c) )
print ( sys.getsizeof(d) )
print ( sys.getsizeof(e) )
print ( sys.getsizeof(f) )
```

Output: (bytes) ...

```
28      # <--- class int ...
24      # <--- class float ...
32      # <--- class complex ...
28      # <--- class boolean ...
65      # <--- class str ...
96      # <--- class list ...
```

Builtin Data Types

Example 4: Formatting data type output ...

```
print("--- a = {:2d} ... ".format(a) );       # <-- Format integer output.
print("--- b = {:.2f} ... ".format(b) );      # <-- two-decimal places
print('--- c = {:.2f}'.format(c))             #     of accuracy.
print("--- d = {:.5s} ... ".format( str(d) ))
print("--- e = {:15s} ... ".format(e) )
output = ["%.5s" % elem for elem in f ]       # <-- convert list to string ...
print("--- f = ", output )
```

Output:

```
--- a =  1 ...
--- b = 1.50 ...
--- c = 1.00+1.50j
--- d = True ...
--- e = this is a string ...
--- f =  ['A', 'B', 'C', 'D']
```

Floating-Point Numbers

Definition. Floating point variables and constants are used represent values outside of the integer range (e.g., `3.4`, `-45.33` and `2.714`) and are either very large or small in magnitude, (e.g., `3.0e-25`, `4.5e+05`, and `2.34567890098e+19`).

IEEE 754 Floating-Point Standard. Specifies that a floating point number take the form:

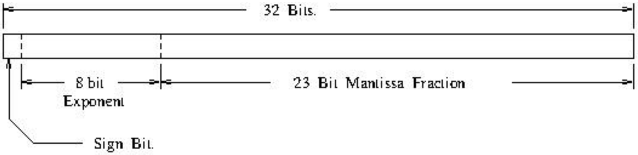
$$X = \sigma \cdot m \cdot 2^{E}. \tag{1}$$

Here:

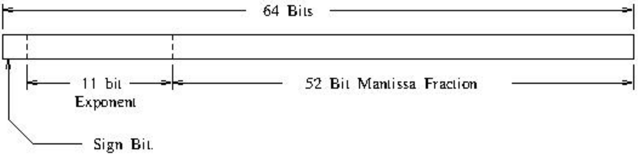
- σ represents the sign of the number.
- m is the mantissa (interpreted as a fraction $0 < m < 1$).
- E is the exponent.

IEEE 754 Floating-Point Standard

Ensures floating point implementions and arithmetic are consistent across various types of computers (e.g., PC and Mac).



IEEE FLOATING POINT ARITHMETIC STANDARD FOR 32 BIT WORDS.



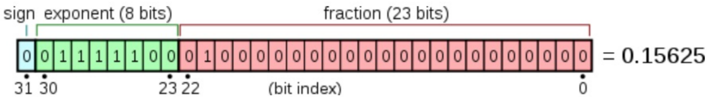
IEEE FLOATING POINT ARITHMETIC STANDARD FOR DOUBLE PRECISION FLOATS.

Largest and Smallest Floating-Point Numbers

```
====================================================================
                  Default
Type   Contains     Value   Size      Range and Precision
====================================================================

float  IEEE 754       0.0   32 bits   +- 13.40282347E+38 /
       floating point                 +- 11.40239846E-45

       Floating point numbers are represented to approximately
       6 to 7 decimal places of accuracy.

double IEEE 754       0.0   64 bits   +- 11.79769313486231570E+308 /
       floating point                 +- 14.94065645841246544E-324

       Double precision numbers are represented to approximately
       15 to 16 decimal places of accuracy.
====================================================================
```

Working with Double Precision Numbers

Simple Example. Here is the floating point representation for 0.15625



Note. Keep in mind that floating-point numbers are stored in a binary format – this can lead to surprises.

For example, when the decimal fraction 1/10 (0.10 in base 10) is converted to binary, the result is an expansion of infinte length.

Bottom line: You cannot store 0.10 precisely in a computer.

Working with Double Precision Numbers

Accessing the Math Library Module

```
import math;  # <-- import the math library ...
```

Math Constants

```
Method          Description
==========================================================================
math.e          Returns Euler's number (2.7182 ...).

math.inf        Returns floating-point positive infinity.

math.pi         Returns PI (3.1415926 ...).
==========================================================================
```

Math Methods

```
Method          Description
==========================================================================
math.acos()     Returns the arc cosine of a number.
math.acosh()    Returns the inverse hyperbolic cosine of a number.
math.asin()     Returns the arc sine of a number.
math.asinh()    Returns the inverse hyperbolic sine of a number.
==========================================================================
```

Working with Double Precision Numbers

Math Methods (continued) ...

```
Method           Description
===================================================================
math.atan()      Returns the arc tangent of a number in radians
math.atan2()     Returns the arc tangent of y/x in radians
math.ceil()      Rounds a number up to the nearest integer
math.cos()       Returns the cosine of a number
math.cosh()      Returns the hyperbolic cosine of a number
math.exp()       Returns E raised to the power of x
math.fabs()      Returns the absolute value of a number
math.floor()     Rounds a number down to the nearest integer
math.gcd()       Returns the greatest common divisor of two integers
math.isfinite()  Checks whether a number is finite or not
math.isinf()     Checks whether a number is infinite or not
math.isnan()     Checks whether a value is NaN (not a number) or not
math.isqrt()     Rounds a square root number down to the nearest integer
math.ldexp()     Returns the inverse of math.frexp() which is
                 x * (2**i) of the given numbers x and i
math.lgamma()    Returns the log gamma value of x
===================================================================
```

Working with Double Precision Numbers

Math Methods (continued) ...

```
Method              Description
====================================================================
math.log()          Returns the natural logarithm of a number, or the
                    logarithm of number to base.
math.log10()        Returns the base-10 logarithm of x
math.log1p()        Returns the natural logarithm of 1+x
math.log2()         Returns the base-2 logarithm of x
math.perm()         Returns the number of ways to choose k items from n
                    items with order and without repetition
math.pow()          Returns the value of x to the power of y
math.prod()         Returns the product of all the elements in an iterable
math.radians()      Converts a degree value into radians
math.remainder()Returns the closest value that can make numerator
                    completely divisible by the denominator
math.sin()          Returns the sine of a number
math.sinh()         Returns the hyperbolic sine of a number
math.sqrt()         Returns the square root of a number
math.tan()          Returns the tangent of a number
math.tanh()         Returns the hyperbolic tangent of a number
math.trunc()        Returns the truncated integer parts of a number
====================================================================
```

Working with Double Precision Numbers

Example 4: Formatting PI ...

```
import math;       # <-- import math library.
PI = math.pi;      # <-- create user-defined constant.

print("--- PI = {:.2f} ...".format(PI) );  # <-- 2 decimal places.
print("--- PI = {:.15f} ...".format(PI) ); # <-- 15 decimal places.
print("--- PI = {:8.2f} ...".format(PI) ); # <-- 8 characters wide,
                                           #     2 decimal places.
print("--- PI = {:16.12f} ...".format(PI) );# <-- 16 characters wide,
                                            #     12 decimal places.
print("--- PI = {:16.6e} ...".format(PI) ); # <-- exponential format.
```

Output:

```
--- PI = 3.14 ...
--- PI = 3.141592653589793 ...
--- PI =     3.14 ...
--- PI =   3.141592653590 ...
--- PI =     3.141593e+00 ...
```

Variables

Working with Variables

Definition. A variable is a placeholder name for any number or unknown.

Assignment Statements. The equality sign is used to assign values to variables:

```
>>> x = 3
>>> print(x)
3
>>>
```

Variable Names. Here are the rules:

- Can be assigned to scalars, vectors and matrices.
- A mixture of letters, digits, and the underscore character. The first character in a variable name must be a letter.

Working with Variables

More than one command may be entered on a single line if the commands are separated by commas or semicolons.

```
>>> x = 3; y = 4
>>> print( x, y)
3 4
>>>
```

Comment Statements

The # symbol indicates the beginning of a comment and, as such, the Python interpreter will disregard the rest of the command line.

Arithmetic Expressions

Arithmetic Operators and Expressions

```
Meaning Of Arithmetic Operators
===================================================================
Operator    Meaning                                   Example
===================================================================
      **    Exponentiation of "a" raised to the     2**3 = 2*2*2 = 8
            power of "b".
       *    Multiply "a" times "b".                 2*3 = 6
       /    Right division (a/b) of "a" and "b".    2/3 = 0.6667
       +    Addition of "a" and "b"                 2 + 3 =  5
       -    Subtraction of "a" and "b"              2 - 3 = -1
===================================================================
```

Here are three examples:

```
>>> 2+3   # Compute the sum "2" plus "3"
5
>>> 3*4   # Compute the product "3" times "4"
12
>>> 4**2;  # Compute "4" raised to the power of "2"
16
```

Rules for Evaluation of Arithmetic Expressions

Rules for Evaluation:

- Operators having the highest precedence are evaluated first.
- Operators of equal precedence are evaluated left to right.

Example. The expression

```
>> 2+3*4**2
```

evaluates to 50. That is:

```
    2 + 3*4**2    <== exponent has the highest precedence.
==> 2 + 3*16      <== then multiplication operator.
==> 2 + 48        <== then addition operator.
==> 50
```

Precedence of Arithmetic Operators

Parentheses may be used to alter the order of evaluation.

```
Precedence Of Arithmetic Expressions
===================================================================
Operators Precedence                                        Comment
===================================================================
   ()          1     Innermost parentheses are evaluated first.
   **          2     Exponentiation operations are evaluated
                     right to left.
  * /          3     Multiplication and right division opera-
                     tions are evaluated left to right.
  + -          4     Addition and subtraction operations are
                     evaluated left to right.
===================================================================
```

Precedence of Arithmetic Operators

Example 1. The expression

```
>> (2 + 3*4**2)/2
```

generates ans = 25. That is,

```
    (2 + 3*4**2)/2   <== evaluate expression within
                         parentheses. Exponent has
                         highest precedence.
==> (2 + 3*16)/2     <== then multiplication operator.
==> (2 + 48)/2       <== then addition operator inside
                         parentheses.
==> (50)/2           <== then division operator
==> 25
```

Precedence of Arithmetic Operators

Example 2. Parentheses are also used in function calls, e.g.,

```
>> 4.0*math.sin( math.pi/4 + math.pi/4 )
```

The order of evaluation is as follows:

```
    4*math.sin( math.pi/4 + math.pi/4 ) <== begin evaluation of left-h
                                            side multiplication.
==> 4*math.sin( math.pi/4 + math.pi/4 ) <== evaluate expression within
                                            function parentheses, star
                                            with leftmost division.
==> 4*math.sin( 0.7854 + pi/4 )   <== evaluate right-hand side
                                     division.
==> 4*math.sin( 0.7854 + 0.7854 ) <== evaluate sum.
==> 4*math.sin( 1.5708 )          <== sin(pi) function call.
==> 4*1.0                         <== finish evaluation of left-hand
                                      side multiplication.
==> 4.0
```

Precedence of Arithmetic Operators

Example 3. Verify that

$$sin(x)^2 + cos(x)^2 = 1.0 \tag{2}$$

for some arbitrary values of `x`. The Python code is

```
>>> x = math.pi/3;
>>> print( math.sin(x)**2 + math.cos(x)**2 - 1.0 )
0.0
>>>
```

Order of Evaluation: (1) $sin(x)$, (2) $sin(x)^2$, (3) $cos(x)$, (4) $cos(x)^2$, (5) addition, (6) subtraction.

Modulo Operator

Definition

The modulo operator (%) returns the remainder of dividing two numbers (the term modulo comes from a branch of mathematics called modular arithmetic). It shares the same level of precedence as the multipliction and division operators.

Examples:

```
5  % 2 ==> 2 * 2 + 1 ==> 1.
3 * 4 % 5 ==> 12 % 5 ==> 2 * 5 + 2 ==> 2.
```

Modulo Operator with int

```
>>> 15 % 4
3
>>> 10 % 16
10
```

Modulo Operator

Modulo Operator with floats

The modular operator used with a float returns the remainder of division as a float.

Example:

```
12.4 % 2.5 ==> 4 * 2.5 + 2.4 ==> 2.4.
```

Modulo Operator with floats

```
>>> import math
>>> print( math.fmod ( 12.4, 2.5 ) )
2.4
>>>
```

Handling Numerical Errors Gracefully

Simulate and Catch Divide-by-zero Error Condition

```
x = 0.0; y = 3.6; z = 5.0;
print("--- x = {:.2f}, y = {:.2f}, z = {:.2f}  ... ".format(x,y,z) );

try:
   result = y / x;
   print("--- Division: y / x --> {:.2f} ... ".format(result) );
except ZeroDivisionError:
   print("--- Division: y / x --> Error: divide by zero ... ");
```

Output:

```
--- x = 0.00, y = 3.60, z = 5.00  ...
--- Division: y / x --> Error: divide by zero ...
```

Handling Numerical Errors Gracefully

Simulate and Catch Numerical Overflow Error Condition

```
i=1
f = 3.0**i
for i in range(10):
    print("--- i = {:3d}, f = {:.2e} ".format(i,f) );
    try:
        f = f ** 2
    except OverflowError as err:
        print("--- Numerical Overflow error ... ");
```

Abbreviated Output:

```
--- i =   0, f = 3.00e+00
--- i =   1, f = 9.00e+00
--- i =   2, f = 8.10e+01
--- i =   3, f = 6.56e+03
--- i =   4, f = 4.30e+07
--- i =   5, f = 1.85e+15
--- i =   6, f = 3.43e+30
--- i =   7, f = 1.18e+61
--- i =   8, f = 1.39e+122
--- i =   9, f = 1.93e+244
--- Numerical Overflow error ...
```

Program Control

Program Control

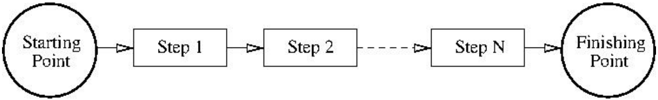
Behavior models coordinate a set of what we will call steps. Two questions need to be answered at each step:

- When should each step be taken?
- When are the inputs to each step determined?

Abstractions that allow for the ordering of functions include:

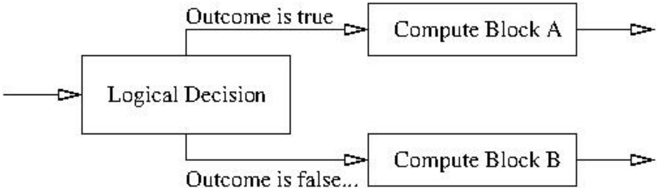
- Sequence constructs,
- Branching constructs,
- Repetition/looping constructs,

Sequences:

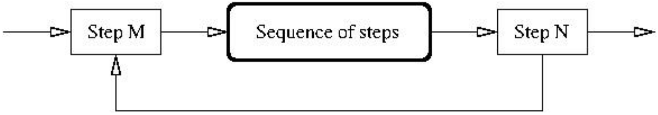


Program Control Abstractions

Selection Constructs:



Looping Constructs:



Control Structures

Definition

A control structure directs the order of execution of statements in a program – this sequence is referred to as the program's control of flow.

Table of Relational Operators:

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

Relational Operators

Example 1: Evaluation of relational operators:

```
x = 4; y = 5; z = 6
print("--- x = {:2d}, y = {:2d}, z = {:2d} ...".format(x,y,z))
print('--- x > y   is', x > y )
print('--- x >= y  is', x >= y )
print('--- x < y   is', x < y )
print('--- x <= y  is', x <= y )
print('--- x == y   is', x == y )
print('--- x != y   is', x != y )
```

Output:

```
--- x =  4, y =  5, z =  6 ...
--- x > y   is False
--- x >= y  is False
--- x < y   is True
--- x <= y  is True
--- x == y  is False
--- x != y  is True
```

Boolean Operators

Boolean **And** Operator ....

A	**B**	**A and B**
True	True	True
True	False	False
False	True	False
False	False	False

Boolean **Or** Operator ....

A	**B**	**A or B**
True	True	True
True	False	True
False	True	True
False	False	False

Boolean **Not** Operator ....

A	**Not A**
True	False
False	True

Boolean Operators

Example 2: Evaluate logical expressions.

```
a = True; b = False

print("--- a and b is {:s} ...".format(str( a and b )))
print("--- a or b  is {:s} ...".format(str( a or b )))
print("--- not a   is {:s} ...".format(str( not a )))
```

Output:

```
--- a and b is False ...
--- a or b  is True ...
--- not a   is False ...
```

Compound Expressions

Example 3: Evaluate compound expressions.

```
x = 4; y = 5; z = 6
print("--- x > y and y <= z --> {:s} ...".format(str( x > y and y <= z )))
print("--- x >= y or y <= z --> {:s} ...".format(str( x >= y or y <= z )))
```

Output:

```
--- x > y and y <= z --> False ...
--- x >= y or y <= z --> True ...
```

Branching Constructs

Syntax for if, else and elif:

```
if <condition>:
   statement 1;
   statement 2;
   statement 3;

   statement 4;
```

```
if <condition>:
   statement 1;
   statement 2;
else:
   statement 3;
   statement 4;
```

```
if <condition>:
   statement 1;
elif <condition>:
   statement 2;
elif <condition>:
   statement 3;
else:
   statement 4;
```

Key Points:

- Left: Statements 1-4 will be executed when the condition (can be a value, variable, or expression) evaluates to True.
- Middle: Statements 1-2 will execute when condition evaluates to True. Otherwise, statements 3-4 will execute.
- Right: The elif (i.e., else-if) statement chains a series of conditional statements.

Branching Constructs

Example 1: Exercise if-else statement ...

```
for i in range(1, 5):
   if i%2 == 1:
      print("--- i = {:3d} --> odd number ...".format(i) );
   else:
      print("--- i = {:3d} --> even number ...".format(i) );
```

Output:

```
--- i =   1 --> odd number ...
--- i =   2 --> even number ...
--- i =   3 --> odd number ...
--- i =   4 --> even number ...
```

Branching Constructs

Example 2: Exercise if-elif-else statement ...

```
for age in range(2, 21, 2):
    if age <= 5:
       print("--- age = {:3d} --> too young for school ...".format(age) );
    elif age > 5 and age < 10:
       print("--- age = {:3d} --> elementary school ...".format(age) );
    elif age >= 10 and age < 14:
       print("--- age = {:3d} --> middle school ...".format(age) );
    elif age >= 14 and age <= 18:
       print("--- age = {:3d} --> high school ...".format(age) );
    else:
       print("--- age = {:3d} --> tertiary education ...".format(age) );
```

Abbreviated Output:

```
--- age =   2 --> too young for school ...
--- age =   4 --> too young for school ...
--- age =   6 --> elementary school ...
--- age =   8 --> elementary school ...
--- age =  10 --> middle school ...
...
--- age =  20 --> tertiary education ...
```

Looping Constructs

Syntax for while and for loops

```
while <condition>:          for value in sequence:
   statement(s);               statement(s);
```

Key Points:

- A while loop will execute statement(s) as long as a condition is true.
- If the condition expression involves a counter variable i, remember to increment, otherwise the loop will continue forever.
- A break statement can stop a loop even while the condition is true. A continue statement can stop the current iteration and continue with the next
- For loops iterate over a sequence (e.g., list, dictionary, set).

Looping Constructs

Example 1: Simple while loop.

```
Python Code                                  Program Output
============================================  ====================
i = 1                                         --- i = 1.00 ...
while i <= 10:                                --- i = 3.00 ...
   print("--- i = {:.2f} ...".format(i) )     --- i = 5.00 ...
   i = i + 2                                  --- i = 7.00 ...
                                              --- i = 9.00 ...
```

Example 2: Simple while loop with break statement.

```
Python Code                                   Program Output
=============================================  ====================
i = 1                                          ---  i = 1.00 ...
while i <= 10:                                 ---  i = 3.00 ...
   print("---  i = {:.2f} ...".format(i) )     ---  i = 5.00 ...
   if i == 5:
      break
   i = i + 2
```

Looping Constructs

Example 3: Simple while loop with continue statement.

```
Python Code                                      Program Output
==============================================   ======================
i = 1                                            ---  i =  1.00 ...
while i <= 10:                                   ---  i =  3.00 ...
   print("---  i = {:5.2f} ...".format(i) )      ---  i =  5.00 ...
   if i == 5:                                    ---  i =  6.00 ...
      i = i + 1                                  ---  i =  8.00 ...
      continue                                   ---  i = 10.00 ...
   i = i + 2
```

Example 4: While loop with else condition ...

```
Python Code                                   Program Output
===========================================   ======================
i = 1                                         ---  i = 1.00 ...
while i < 6:                                  ---  i = 2.00 ...
   print("---  i = {:.2f} ...".format(i) )    ---  i = 3.00 ...
   i += 1                                     ---  i = 4.00 ...
else:                                         ---  i = 5.00 ...
   print("---  i no longer less than 6")      ---  i no longer less than 6
```

Looping Constructs

Example 5: Use for loop to traverse list of cars ...

```
Python Code                                            Program Output
=================================================      ======================
cars = ['Toyota', 'Honda', 'BMW', 'Tesla']             --- car 0: Toyota ...
for i in range(len(cars)):                             --- car 1: Honda ...
    print("--- car {:d}: {:s} ...".format(i,cars[i]))  --- car 2: BMW ...
                                                       --- car 3: Tesla ...
```

Example 6: Array generated by np.linspace(0,10,num=11) ...

```
Python Code                                          Program Output
=================================================    ======================
coords = np.linspace(0,10,num=11)                    ---  x( 0) =  0.00 ...
i = 0                                                ---  x( 1) =  1.00 ...
for ii in coords:                                    ---  x( 2) =  2.00 ...
   print("--- x({:2d}) = {:5.2f} ...".format(i, ii)) ---  ...
   i = i + 1                                         ---  x(10) = 10.00 ...
```

Looping Constructs

Example 7: Use nested for loop (adjective, fruit) pairs ...

```
Python Code
================================================
adjective = [ "red", "big", "tasty", "spoiled" ]
fruits    = ["apple", "banana", "cherry"]

for x in adjective:
   for y in fruits:
       print("--- {:s} {:s} ...".format(x, y) )
```

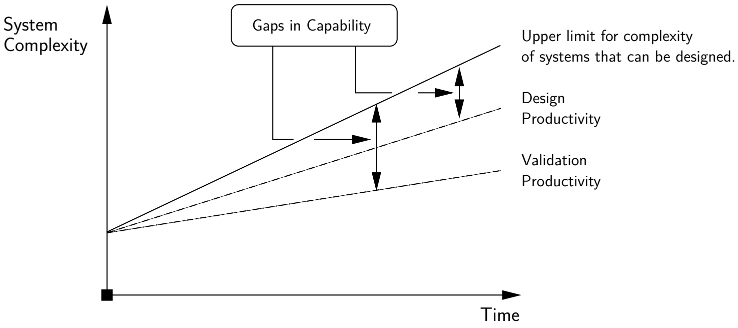
```
Program Output
=====================
--- red apple ...
--- red banana ...
--- red cherry ...
--- big apple ...
--- big banana ...
--- big cherry ...
--- tasty apple ...
--- tasty banana ...
--- tasty cherry ...
--- spoiled apple ...
--- spoiled banana ...
--- spoiled cherry ...
```

Functions

Functions: Strategies for Handling Complexity

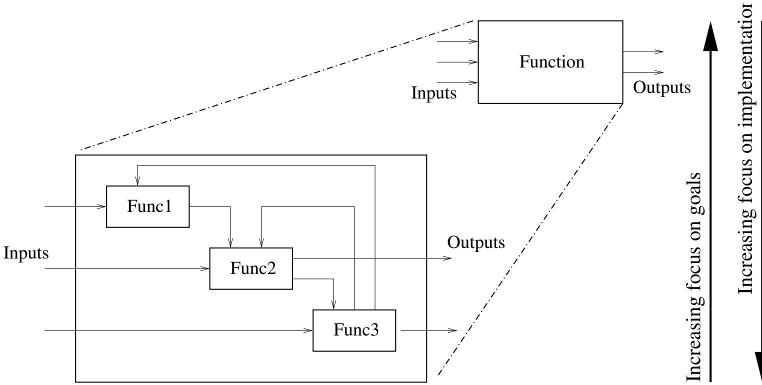
Productivity Concerns

System designers and software developers need to find ways of being more productive, just to keep the duration and economics of design development in check.



Functions: Strategies for Handling Complexity

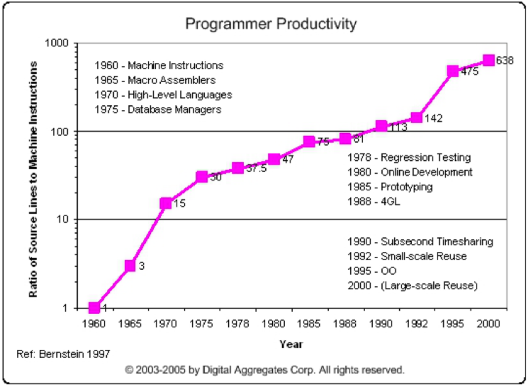
Simplify models of funtionality by decomposing high-level functions into networks of lower-level functionality:

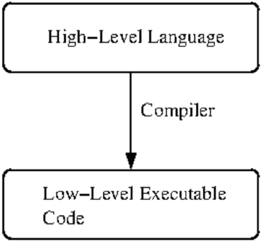


Functions: Strategies for Handling Complexity

Create High-Level Description of Solution:

Increasing System Complexity: Software programmers need to find ways to solve problems at high levels of abstraction.





Python: Builtin Functions

Built-in Functions

A
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Python: Builtin Functions

Example 1: abs() returns the absolute value of a number.

```
>>> print ( abs( -15 ) )
15
>>>
```

Example 2: max() and min() return the maximum/minimum value in a list.

```
>>> a = [ -3, 2, 5, -10, 12, -14 ]
>>> print ( max( a ) )
12
>>> print ( min( a ) )
-14
>>> print("--- range = {:2d} ...".format( max(a) - min(a) ))
--- range = 26 ...
>>>
```

Python: User-Defined Functions

User-defined Functions

User-defined functions are defined using the def keyword. Information can be passed to functions as arguments. Functions have the option of returning one or more values.

Example 1: Let's create a simple welcome message.

```
def WelcomeMessage():
    print("--- Welcome !! ... ");
```

Calling the Function:

```
>>> WelcomeMessage()
--- Welcome !! ...
>>>
```

Python: User-Defined Functions

Example 2: Function with two arguments (passed to the function as a comma-separated list after the function name).

```
def print_name02(firstName, familyName ):
    print("---    Name:" + firstName + " " + familyName )
```

Calling the Function:

```
print_name02(  "Bart", "Simpson");
print_name02(  firstName =    "Bart", familyName = "Simpson");
print_name02( familyName = "Simpson",  firstName = "Bart" );
```

Output:

```
---    Name:Bart Simpson
---    Name:Bart Simpson
---    Name:Bart Simpson
```

Python: User-Defined Functions

Example 3: Function to return square of argument value ...

```
def my_square_function(x):
    return x * x
```

Calling the Function:

```
x = 2.0;
print("--- Input: {:.2f} --> squared: {:5.2f} ...".format( x,
                                                       my_square_function(x)))
x = 3.0;
print("--- Input: {:.2f} --> squared: {:5.2f} ...".format( x,
                                                       my_square_function(x)))
```

Output:

```
--- Input: 2.00 --> squared:  4.00 ...
--- Input: 3.00 --> squared:  9.00 ...
```

First Program

(Evaluate and Plot Sigmoid Function)

Problem Desription

Problem Description

In neural network models, the sigmoid function:

$$\sigma(x) = \left[\frac{1}{1 + e^{-x}} \right]. \quad (3)$$

serves as an activation. Our first program evaluates and plots $\sigma(x)$ over the range $x \in [-10, 10]$.

Running the Program

From the terminal window, simply type:

```
prompt >> python3 TestSigmoidFunction.py
```

Evaluate and Plot Sigmoid Function

The Python interpreter/compiler will complain if one or more of the required packages (e.g., matplotlib) are missing.

Use pip to install missing Python Packages

The standard package-management system used to install and manage software packages is called pip (or pip3).

Example: And installation is easy!

```
prompt >> pip3 install numpy
prompt >> pip3 install matplotlib
```

To get a list of installed packages:

```
prompt >> pip3 list
```

Evaluate and Plot Sigmoid Function

Abbreviated Output:

```
Package                      Version
---------------------- ---------
....
jupyter                      1.0.0
Keras                        2.4.3
....
matplotlib                   3.4.1
....
numpy                        1.19.5
....
pandas                       1.1.5
....
scikit-learn                 0.24.2
scipy                        1.6.2
....
sklearn                      0.0
```

Program Source Code in Visual Studio Code

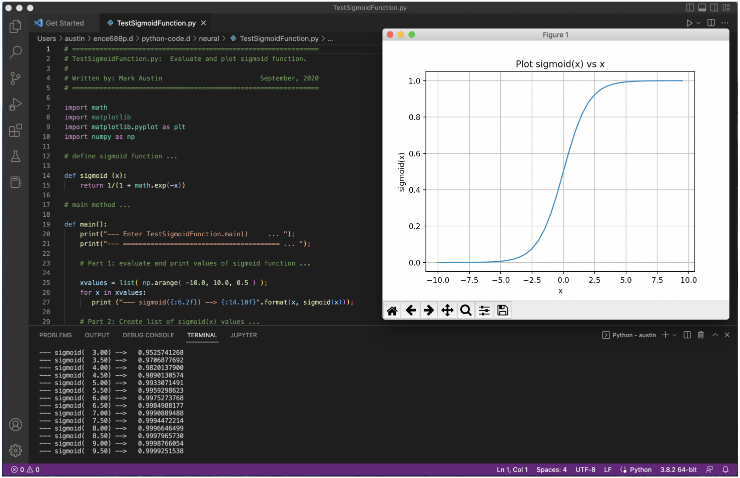
TestSigmoidFunction.py

Users > austin > ence688p.d > python-code.d > neural > TestSigmoidFunction.py > ...

```python
# ===============================================================
# TestSigmoidFunction.py:  Evaluate and plot sigmoid function.
#
# Written by: Mark Austin                        September, 2020
# ===============================================================

import math
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# define sigmoid function ...

def sigmoid (x):
    return 1/(1 + math.exp(-x))

# main method ...

def main():
    print("--- Enter TestSigmoidFunction.main()     ... ");
    print("--- ======================================== ... ");

    # Part 1: evaluate and print values of sigmoid function ...

    xvalues = list( np.arange( -10.0, 10.0, 0.5 ) );
    for x in xvalues:
        print ("--- sigmoid({:6.2f}) --> {:14.10f}".format(x, sigmoid(x)));

    # Part 2: Create list of sigmoid(x) values ...

    yvalues = []
    for x in xvalues:
        yvalues.append( sigmoid(x) );

    # Part 3: Organize and display plot ...

    fig, ax = plt.subplots()
    ax.plot( xvalues, yvalues )
    ax.set(xlabel='x', ylabel='sigmoid(x)', title='Plot sigmoid(x) vs x')
    ax.grid()

    # display plot ...
```

Ln 1, Col 1 Spaces: 4 UTF-8 LF Python 3.8.2 64-bit

Program Source Code + Output in Visual Studio Code



Program Source Code

```
# ===========================================================
# TestSigmoidFunction.py: Evaluate/plot sigmoid function.
#
# Written by: Mark Austin                     September, 2020
# ===========================================================

import math
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# define sigmoid function ...

def sigmoid (x):
    return 1/(1 + math.exp(-x))

# main method ...

def main():
    print("--- Enter TestSigmoidFunction.main() ...");
    print("--- ================================ ...");

    # Part 1: Evaluate and print sigmoid function

    xvalues = list( np.arange( -10.0, 10.0, 0.5 ) );
    for x in xvalues:
       print ("--- sigmoid({:6.2f}) --> {:14.10f}".format(x, sigmoid(x)));

    # Part 2: Create list of sigmoid(x) values ...
```

Program Source Code

```
    # Part 2: Create list of sigmoid(x) values ...

    yvalues = []
    for x in xvalues:
       yvalues.append( sigmoid(x) );

    # Part 3: Organize and display plot ...

    fig, ax = plt.subplots()
    ax.plot( xvalues, yvalues )
    ax.set(xlabel='x', ylabel='sigmoid(x)',
           title='Plot sigmoid(x) vs x')
    ax.grid()

    # display and save plot ...

    plt.show()

    fig.savefig("sigmoid-plot.jpg")

    print("--- ================================ ...");
    print("--- Leave TestSigmoidFunction.main() ...");

# call the main method ...

main()
```

Program Source Code

Points to Note:

- Line comment statements begin with the # character.
- Lines 7-10 import the math, matplotlib, matplotlib.pyplot and numpy modules to the program, and make the functions therein available.
- Functions are the primary method of code organization and reuse in Python.
- User-defined functions are declared with the `def` keyword. A function contains a block of code with an optional `return` keyword.
- Lines 13-14 evaluate and return the sigmoid function.
- Use of the second function, `main()`, is a carry over from programming with C, where all programs begin their execution in `main()`. Its use in Python is optional.

Program Source Code

Points to Note (continued):

- Line 25 creates a list of x coordinates. The numpy function `np.arange()` covers $[-10, 10]$ in increments of 0.5.
- Lines 26-27 systematically traverse the elements of `xvalues`, and compute and print the corresponding values of the `sigmoid()` function.
- Line 27 formats and prints the output. The specification `{:6.2}f` means that the output should be printed as a floating point number, six characters wide, and with two decimal places of accuracy to the right of the decimal point.
- Lines 31-33 traverse the elements of `xvalues`, and systematically assemble a list of sigmoid function `yvalues`.
- Lines 37-47 format a plot of `yvalues` vs `xvalues`, and save to `sigmoid-plot.jpg`.

Builtin Containers and Collections

(Working with Lists, Dictionaries, Sets)

Builtin Containers and Collection

Containers and Collections

A container is an object that stores objects, and provides a way to access and iterate over them. Collections are container data types, namely lists, sets, tuples, dictionary.

Builtin Collection Data Types:

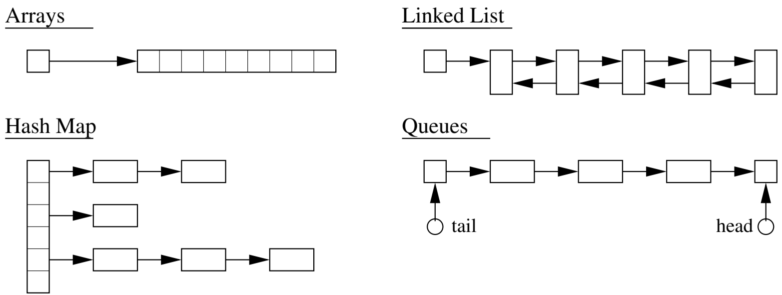
- **List:** A list is a collection which is ordered and changeable.
- **Dictionary:** A dictionary is a collection which is ordered and changeable. No duplicate members.
- **Set:** A set is a collection which is unordered, unchangeable and unindexed. No duplicate members.
- **Tuple:** A tuple is a collection which is ordered and unchangeable.

Working with Lists

List

Lists are used to store multiple items in a single variable. A list may store multiple types (heterogeneous) of elements.

Array, List, HashMap, and Queue Structures



Working with Lists

Basic List Methods

```
Method        Description
=========================================================================
append()      Adds an element at the end of the list
clear()       Removes all the elements from the list
copy()        Returns a copy of the list
count()       Returns the number of elements with the specified value
extend()      Add the elements of a list (or any iterable), to the end of
              the current list.
index()       Returns the index of the first element with the
              specified value.
insert()      Adds an element at the specified position.
remove()      Removes the item with the specified value.
reverse()     Reverses the order of the list.
sort()        Sorts the list.
=========================================================================
```

Working with Lists

Example 1: Create working lists ...

```
  list01 = [ "apple", "orange", "avocado", "banana", "grape", "watermelon"]
  list02 = [ "apple", "avocado", "banana", "banana", "grape", "watermelon"]

  print ("--- List01: %s ..." %( list01 ))
  print ("--- List02: %s ..." %( list02 ))

# Create list with mix of data types ...

  list03 = [ "apple", 40, True, 2.5 ]

  print ("--- List03 (with multiple data types): %s ..." %( list03 ))
```

Output:

```
--- List01: ['apple', 'orange', 'avocado', 'banana', 'grape', 'watermelon'] ...
--- List02: ['apple', 'avocado', 'banana', 'banana', 'grape', 'watermelon'] ...

--- List03 (with multiple data types): ['apple', 40, True, 2.5] ...
```

Working with Lists

Example 2: Access list items ...

```
list04 = list(( "apple", 40, True, 2.5, False ))

print ("---   list04[0]: %s ..." %( list04[0] ))
print ("---   list04[1]: %s ..." %( list04[1] ))
print ("---   list04[2]: %s ..." %( list04[2] ))
print ("---   list04[3]: %s ..." %( list04[3] ))
print ("---   list04[4]: %s ..." %( list04[4] ))
```

Output:

```
---   list04[0]: apple ...
---   list04[1]: 40 ...
---   list04[2]: True ...
---   list04[3]: 2.5 ...
---   list04[4]: False ...
```

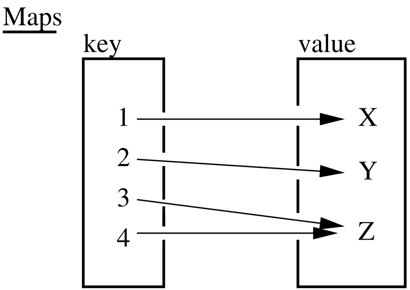
Source Code: See: python-code.d/collections/

Working with Dictionaries

Dictionary

Dictionaries store data values as key:value pairs. As of Python 3.7, a dictionary is a collection which is ordered, changeable and do not allow duplicates.

Key:Value Map Operations



Working with Dictionaries

Basic Dictionary Methods

```
Method       Description
========================================================================
clear()      Removes all the elements from the dictionary.
copy()       Returns a copy of the dictionary.
fromkeys()   Returns a dictionary with the specified keys and value.
get()        Returns the value of the specified key.
items()      Returns a list containing a tuple for each key value pair.
keys()       Returns a list containing the dictionary's keys.
pop()        Removes the element with the specified key.
popitem()    Removes the last inserted key-value pair.
update()     Updates the dictionary with the specified key-value pairs.
values()     Returns a list of all the values in the dictionary.
========================================================================
```

Working with Dictionaries

Example 1: Create dictionary of car attributes.

```
car01 = {  "brand": "Honda",         # <-- Create simple dictionary ....
           "model": "Acura",
           "miles": 25000,
             "new": False,
            "year": 2016
        }

print ("--- Car01: %s ..." %( car01 ))  # <-- print dictionary ...
```

Output: Print simple dictionary.

```
--- Car01: {'brand': 'Honda', 'model': 'Acura',
            'miles': 25000, 'new': False, 'year': 2016} ...
```

Working with Dictionaries

Example 2: Systematically access items in Car01 ...

```
print ("--- Car01: brand --> %s ..." %( car01.get("brand") ))
print ("---      : model --> %s ..." %( car01.get("model") ))
print ("---      : miles --> %d ..." %( car01.get("miles") ))
print ("---      : new   --> %s ..." %( car01.get("new") ))
print ("---      : year  --> %d ..." %( car01.get("year") ))
```

Output:

```
--- Access items in Car01 ...
--- Car01: brand --> Honda ...
---      : model --> Acura ...
---      : miles --> 25000 ...
---      : new   --> False ...
---      : year  --> 2016 ...
```

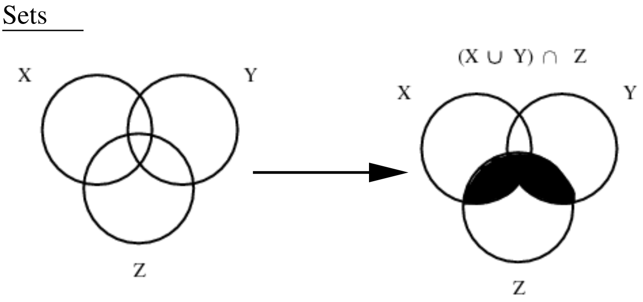
Source Code: See: python-code.d/collections/

Working with Sets

Sets

Sets store multiple items in a single variable. A set is a collection which is unordered, unchangeable (but you can remove items and add new items) and unindexed.

Set Operations



Working with Sets

Basic Set Methods

```
Method          Description
==========================================================================
add()           Adds an element to the set.
clear()         Removes all the elements from the set.
copy()          Returns a copy of the set.
discard()       Remove the specified item.
intersection()  Returns a set, that is the intersection of two other sets.
remove()        Removes the specified element.
union()         Return a set containing the union of sets
update()        Update the set with the union of this set and others.
==========================================================================
```

Working with Sets

Example 1: Create working sets; set operations ...

```
set01 = { 1, 2, 3, 4, 5, 6, 7 }
set02 = { 6, 7, 8, 9, 10 }
set03 = {"apple", "banana", "cherry"}
set04 = {True, False, False}

print ("--- Set01.union(Set02) : %s ..." %( set01.union(set02) ))
print ("--- Set01.intersection(Set02) : %s ..."
                                    %( set01.intersection(set02) ))
```

Output:

```
--- Create working sets ...
--- Set01: {1, 2, 3, 4, 5, 6, 7} ...
--- Set02: {6, 7, 8, 9, 10} ...
--- Set03: {'cherry', 'banana', 'apple'} ...
--- Set04: {False, True} ...

--- Set01.union(Set02) : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ...
--- Set01.intersection(Set02) : {6, 7} ...
```

Working with Sets

Example 2: Add items to set03, then print ...

```
set03.add("strawberry")
set03.add("kiwi")
print ("--- Set03 (appended): ...")
for x in set03:
    print ("---   %s ..." %(x))
```

Output: Set03 appended ...

```
---   cherry ...
---   strawberry ...
---   banana ...
---   kiwi ...
---   apple ...
```

Source Code: See: python-code.d/collections/

Numerical Python

(NumPy)

Numerical Python (NumPy)

Introduction to NumPy

Numerical Python (NumPy) is an open source Python library that contains computational support for n-dimensional array objects, along with mathematical methods to operate on them.

Key Features:

- Create 0-d, 1-d and 2-d arrays. 3-d blocks.
- Operations on array elements (e.g., min, max, sort).
- Operations on arrays (e.g., reshape, stack).
- Compute matrix properties. Solve matrix equations.

Installation

```
prompt >> pip3 install numpy
```

Numerical Data Types in NumpPy

dtype	Variants	Description
int	int8, int16, int32, int64	Integers
uint	uint8, uint16, uint32, uint64	Unsigned integers
bool	bool	Boolean (True or False)
float	float16, float32, float64, float128	Floating-point numbers
complex	complex64, complex128, complex256	Complex-valued floating point numbers

Working with NumPy

Example 1: Create 0-d, 1-d, and 2-d arrays ...

```
a = np.array(101);   # <-- create 0-d array.
print (a)

a = np.array( [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ); # <-- create 1-d array of
print (a)

a = np.array( ["The", "Brown", "Fox"] ); # <-- array of charcter strings.
a = np.append(a, "!")

for i in a:                            # <-- loop over array indices ...
   print(i)
```

Output:

```
101
[ 1  2  3  4  5  6  7  8  9 10]
The
Brown
Fox
!
```

Working with NumPy

Example 2: Print each array element and its index ...

```
# Create array of character strings ...

a = np.array( ["The", "Brown", "Fox", "!"] );

for i,e in enumerate(a):
    print("--- Index: {}, was: {}".format(i, e))
```

Output:

```
--- Index: 0, was: The
--- Index: 1, was: Quick
--- Index: 2, was: Brown
--- Index: 3, was: Fox
--- Index: 4, was: !
```

Working with NumPy

Example 3: Sort array elements ...

```
# Sort array of floating point numbers ...

a = np.array( [ 2.3, 1.0, 4.5, -13.0, 100.0, 43, -15.0, 0.0 ] )
print(a);
print(np.sort(a));

# Sort array of state abbreviations ...

a = np.array( ["MD", "CA", "RI", "UT", "LA", "AL", "WA", "OR", "CO"] )
print(a);
print(np.sort(a))
```

Output:

```
--- Sort array of floating-point numbers ...
[  2.3    1.     4.5 -13.   100.    43.   -15.     0. ]
[-15.  -13.     0.     1.     2.3    4.5   43.   100. ]
--- Sort array of state abbreviations ...
['MD' 'CA' 'RI' 'UT' 'LA' 'AL' 'WA' 'OR' 'CO']
['AL' 'CA' 'CO' 'LA' 'MD' 'OR' 'RI' 'UT' 'WA']
```

Working with NumPy

Example 4: Create two-dimensional array ...

```
c = np.array( [ ( 0, 1, 4, 3, 2), ( 3, 4, 5, 6, 7),
                ( 6, 7, 8, 9,10), ( 9,10,11,12,13) ] );

PrintMatrix("C", c);        # <-- print formatted matrix ....

print("    Min: {}".format(np.min(c)))
print("    Max: {}".format(np.max(c)))
print("    Average: {}".format(np.average(c)))
print("    Max array index: {}".format(np.argmax(c)))
```

Output:

```
Matrix: C
   0.000    1.000    4.000    3.000    2.000
   3.000    4.000    5.000    6.000    7.000
   6.000    7.000    8.000    9.000   10.000
   9.000   10.000   11.000   12.000   13.000

Min: 0               Average: 6.5
Max: 13              Max array index: 19
```

Working with NumPy

Example 5: Create three-dimensional array block ...

```
c = np.array( [ [ ( 0, 1), (3, 4) ], [(5, 6), (7, 8) ] ] );
print(c)
```

Output:

```
[ [ [0 1]
    [3 4] ]

  [ [5 6]
    [7 8] ] ]
```

Working with NumPy

Example 6: Reshape 1-d array ⟶ 2-d matrix ...

```
d1 = np.arange(20);      # <-- create 1-d test array ...
print(d1);

d1 = d1.reshape(4,5);    # <-- reshape to (4x5) array ...
PrintMatrix("(4x5)", d1 );
```

Output:

```
--- 1-d test array:

  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

--- Reshape to (4x5) matrix ...

Matrix: (4x5)
     0.000     1.000     2.000     3.000     4.000
     5.000     6.000     7.000     8.000     9.000
    10.000    11.000    12.000    13.000    14.000
    15.000    16.000    17.000    18.000    19.000
```

Working with NumPy

Example 7: Create horizontal and vertical array stacks ...

```
d1 = np.array( [ ( 0, 1), ( 3, 4) ] );    # <-- create test arrays ...
d2 = np.array( [ ( 5, 6), ( 7, 8) ] );

PrintMatrix("d1", d1 ); PrintMatrix("d2", d2 );

h1 = np.hstack((d1, d2));                  # <-- create horizontal stack ...
PrintMatrix( "np.hstack(d1, d2)", h1 );
h2 = np.vstack((d1, d2));                  # <-- create vertical stack ...
PrintMatrix( "np.vstack(d1, d2)", h2 );
```

Output:

```
Matrix: d1                 Matrix: np.hstack(d1, d2)
   0.000     1.000            0.000     1.000     5.000     6.000
   3.000     4.000            3.000     4.000     7.000     8.000

Matrix: d2                 Matrix: np.vstack(d1, d2)
   5.000     6.000            0.000     1.000
   7.000     8.000            3.000     4.000
                              5.000     6.000
                              7.000     8.000
```

Working with NumPy

Example 8: Exercise np.zeros() and np.eye() ...

```
matrix02 = np.zeros(shape=(4, 4))   # <-- create (4x4) array of zeros.
PrintMatrix("matrix02", matrix02 );

matrix03 = np.eye(4, dtype = float) # <-- create (4x4) identidy matrix.
PrintMatrix("matrix03", matrix03 );
```

Output:

```
Matrix: matrix02
   0.000    0.000    0.000    0.000
   0.000    0.000    0.000    0.000
   0.000    0.000    0.000    0.000
   0.000    0.000    0.000    0.000

Matrix: matrix03
   1.000    0.000    0.000    0.000
   0.000    1.000    0.000    0.000
   0.000    0.000    1.000    0.000
   0.000    0.000    0.000    1.000
```

Working with NumPy

Example 9: Reshape arrays of random numbers

```
matrix06 = np.random.random((20,1));   # <-- create (20x1) array
PrintMatrix("matrix06", matrix06 );    #     of random numbers.

PrintMatrix ( "matrix06 (reshaped)",   # <-- reshape to (10x2).
              matrix06.reshape(10,2) )
```

Abbreviated Output:

```
--- Original (20x1) matrix    --- Reshape to (10x2) matrix ...

Matrix: matrix06              Matrix: matrix06 (reshaped)
   0.326                         0.326    0.459
   0.459                         0.545    0.419
   0.545                         0.537    0.632
   .....                         .....    .....
   0.803                         .....    .....
   0.014                         0.165    0.803
   0.291                         0.014    0.291
```

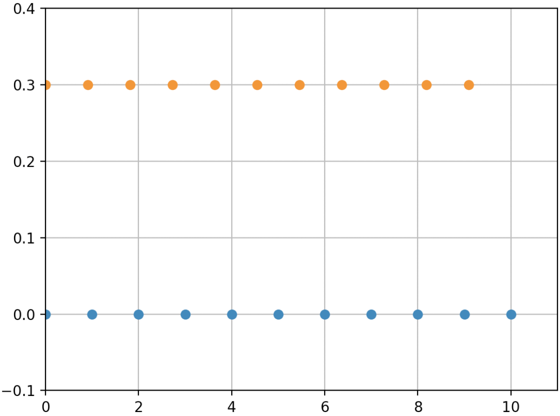
Working with NumPy

Example 10: Generate and plot linear space of coordinates:

```
# ============================================================================
# TestLinspace01.py: Generate arrays of coordinates with np.linspace(), then plot.
# ============================================================================

import numpy as np  # Make numpy available using np.
import matplotlib.pyplot as plt

def main():
    # Generate arrays of x coordinates with np.linspace() ...

    Npoints = 11
    x1 = np.linspace(0, 10, num = Npoints, endpoint=True);
    x2 = np.linspace(0, 10, num = Npoints, endpoint=False);

    # Plot coordinates ...

    y = np.zeros(Npoints)
    plt.plot(x1, y, 'o')
    plt.plot(x2, y + 0.3, 'o')
    plt.ylim( [-0.1, 0.4] )
    plt.xlim( [ 0.0, 11] )
    plt.grid(); plt.show()

# call the main method ...

main()
```

Working with NumPy

Program Output:



Working with NumPy

Example 11: Solve family of matrix equations:

$$\begin{bmatrix} 3 & -6 & 7 \\ 9 & 0 & -5 \\ 5 & -8 & 6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ -4 \end{bmatrix} \quad (4)$$

Part I: Theoretical Considerations:

- A unique solution $\{X\} = [A^{-1}] \cdot \{B\}$ exists when $[A^{-1}]$ exists (i.e., $\det[A] \neq 0$). Expanding det(A) about the first row gives:

$$det(A) = 3\det \begin{bmatrix} 0 & -5 \\ -8 & 6 \end{bmatrix} + 6\det \begin{bmatrix} 9 & -5 \\ 5 & 6 \end{bmatrix} + 7\det \begin{bmatrix} 9 & 0 \\ 5 & -8 \end{bmatrix},$$
$$= 3(0-40) + 6(54+25) + 7(-72-0) = -150. \quad (5)$$

Working with NumPy

Part II: Program Source Code:

```
# ================================================================
# TestMatrixEquations01.py: Compute solution to matrix equations.
#
# Written by: Mark Austin                          November 2022
# ================================================================

import numpy as np
from numpy.linalg import matrix_rank

# Function to print two-dimensional matrices ...

def PrintMatrix(name, a):
    print("Matrix: {:s} ".format(name) );
    for row in a:
        for col in row:
            print("{:8.3f}".format(col), end=" ")
        print("")

# main method ...

def main():
    print("--- Enter TestMatrixEquations01.main()      ... ");
    print("--- ======================================== ... ");

    print("--- Part 1: Create test matrices ... ");
```

Working with NumPy

Part II: Program Source Code: (Continued) ...

```
    A = np.array( [ [ 3, -6,  7],
                    [ 9,  0, -5],
                    [ 5, -8,  6] ]);
    PrintMatrix("A", A);

    B = np.array([ [3], [3], [-4] ]);
    PrintMatrix("B", B);

    print("--- Part 2: Check properties of matrix A ... ");

    rank = matrix_rank(A)
    det  = np.linalg.det(A)

    print("--- Matrix A: rank = {:f}, det = {:f}  ...".format(rank, det) );

    print("--- Part 3: Solve A.x = B ... ");

    x = np.linalg.solve(A, B)
    PrintMatrix("x", x);

    print("--- ======================================== ... ");
    print("--- Leave TestMatrixEquations01.main()      ... ");

# call the main method ...

main()
```

Working with NumPy

Part III: Program Output:

```
# Part 1: Create test matrices ...

Matrix: A
   3.000   -6.000    7.000
   9.000    0.000   -5.000
   5.000   -8.000    6.000

Matrix: B
   3.000
   3.000
  -4.000

# Part 2: Check properties of matrix A ...

Matrix A: rank = 3.000000, det = -150.000000  ...
```

```
# Part 3: Solve A.x = B ...

Matrix: x
   2.000
   4.000
   3.000
```

Tabular Data and Dataset Transformation

(Working with Pandas)

Working with Pandas

Introduction to Pandas

Pandas is an open source Python Library that supports working and analysis of tabular data sets.

Benefits:

- Pandas can clean messy data sets, and make them readable and relevant.
- Pandas allows us to analyze large data sets and make conclusions based on statistical theories.
- Three data structures: Series, DataFrame and Panel.

Installation:

```
prompt >> pip3 install pandas
```

What can Pandas do?

Basic Operations:

- Create series and dataframes.
- Read CSV and JSON files.
- Plot data.

Clean Data:

- Clean empty cells.
- Fix wrong format.
- Remove duplicates.

Advanced Operations:

- Combine (concatenate, join, merge) Panda objects.
- Compute correlations.

Panda Series and DataFrames

Panda Series

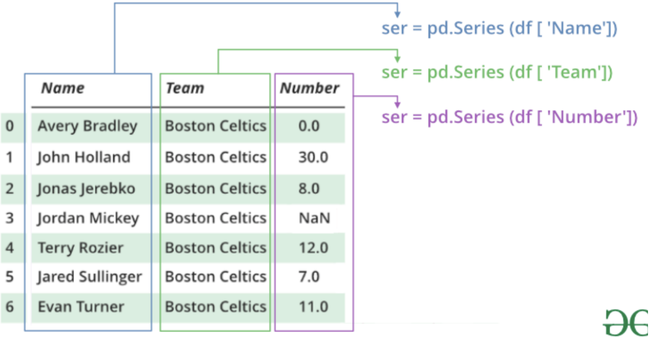
A Panda Series is a one-dimensional ... labeled array capable of holding data of any type (integer, string, float, python objects, etc.).

Panda DataFrame

A Panda DataFrame is a two-dimensional (potentially heterogeneous) tabular data structure with labeled axes for the rows and columns.

Panda Series

Panda Series Elements: columns, data ...



	Name	*Team*	*Number*
0	Avery Bradley	Boston Celtics	0.0
1	John Holland	Boston Celtics	30.0
2	Jonas Jerebko	Boston Celtics	8.0
3	Jordan Mickey	Boston Celtics	NaN
4	Terry Rozier	Boston Celtics	12.0
5	Jared Sullinger	Boston Celtics	7.0
6	Evan Turner	Boston Celtics	11.0

Basic Operations:

- Create a series; access elements; index and select data; binary operations; conversion operations.

Panda Series

Example 1: Manually create series from list:

```
# Part 1: Manually create series ...

a = [1, 2, 3, 4, 3, 2, 1 ]
myvar = pd.Series(a)
print(myvar)

# Part 2: Create series from a list with labels ...

myvar = pd.Series(a, index = ["a", "b", "c", "d", "c", "b", "a" ])
print(myvar)
```

Abbreviated Output: Parts 1 and 2 ...

```
Part 01
0     1
1     2
.....
5     2
6     1
dtype: int64
```

```
Part 02
a     1
b     2
.....
b     2
a     1
dtype: int64
```

Panda Series

Example 2: Manually create series from dictionary:

```
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

Output:

```
day1     420
day2     380
day3     390
dtype: int64
```

Panda Series

Example 3: Create series from NumPy functions

```
# series01 = pd.Series(np.arange(2,8)) ... ");

series01 = pd.Series(np.arange(2,8))
print(series01)
```

Output:

```
0    2
1    3
2    4
3    5
4    6
5    7
dtype: int64
```

Panda Series

Example 4: Create series from NumPy functions

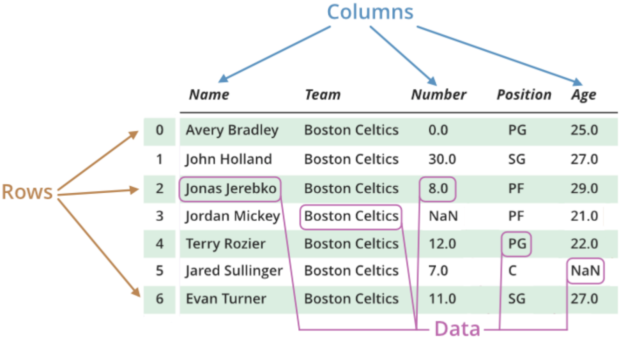
```
series02 = pd.Series( np.linspace(0,10,5) )
print(series02)

print( series02.size)
print( len(series02) )
print( series02.values )
```

Output:

```
0     0.0
1     2.5
2     5.0
3     7.5
4    10.0
dtype: float64

5                         # <-- series02.size ...
5                         # <-- series02 length ...
[ 0.   2.5  5.   7.5 10. ] # <-- series02 values ...
```

Panda DataFrames

Panda DataFrame Elements: rows, columns, data ...



	Name	*Team*	*Number*	*Position*	*Age*
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Basic Operations:

- Create dataframe; deal with rows and columns; index and select data; iterate over rows and columns.

Working with Panda DataFrames

Example 1: Manually create small dataset ...

```
mydataset = {
   'cars': [ "BMW", "Honda", "Acura"],
   'year': [ 2013,     2017,    2022]
}

myvar = pd.DataFrame(mydataset)
print(myvar)
```

Output:

```
     cars  year
0     BMW  2013
1   Honda  2017
2   Acura  2022
```

Working with Panda DataFrames

Example 2: Create dataframes from 1-d and 2-d arrays ...

```
myvar = pd.DataFrame( np.arange(1,8) ) # <-- dataframe from 1-d array
print(myvar)

df = pd.DataFrame( [ [1,2],
                     [3,4],
                     [5,6] ] )         # <-- dataframe from 2-d array
print(df)
```

Abbreviated Output:

```
Dataframe from 1-d np array      Dataframe from 2-d np array
---------------------------      ---------------------------
   0                                    0  1
0  1                                 0  1  2
1  2                                 1  3  4
2  3                                 2  5  6
....
5  6
6  7
```

Working with Panda DataFrames

Example 3: Create simple dataframe from multiple series ...

```
data = {                             # <-- Create dataframe from
   "calories": [520, 480, 400],     #     multiple series.
   "duration": [ 50,  48,  40]
}

myvar = pd.DataFrame(data)
print(myvar)

index = ["day1", "day2", "day3"]  # <-- give each ros a new name.
myvar = pd.DataFrame(data, index)
print(myvar)
```

Output:

```
Part 1: dataframe from series        Part 2: rename rows
     calories  duration                      calories  duration
 0        520        50                day1       520        50
 1        480        48                day2       480        48
 2        400        40                day3       400        40
```

Working with Panda DataFrames

Example 4: Create dataframe from JSON object ...

```
# Create JSON object (same format as Python dictionary) ...

data = {
  "Duration":{ "0":60,  "1":60, "2":60,   "3":45,  "4":45,  "5":60 },
  "Pulse":{    "0":110, "1":117, "2":103, "3":109, "4":117, "5":102 },
  "Maxpulse":{ "0":130, "1":145, "2":135, "3":175, "4":148, "5":127 },
  "Calories":{ "0":409, "1":479, "2":340, "3":282, "4":406, "5":300 }
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
    Duration  Pulse  Maxpulse  Calories
 0        60    110       130       409
 1        60    117       145       479
 2        60    103       135       340
 3        45    109       175       282
 4        45    117       148       406
 5        60    102       127       300
```

Working with Panda DataFrames

Example 5: Select rows and columns from dataframe ...

```
# Select columns of a dataframe ...

print( df[ [ 'Duration','Calories'] ].head() )

# Selecting rows of a dataframe ...

print( df.loc['1'].head() )      # <-- extract and print row 1
print( df.loc['2'].head() )      # <-- extract and print row 2
```

Output:

```
Columns of dataframe       Row 1                  Row 2
--------------------       -----------------      -----------------

    Duration  Calories     Duration      60       Duration      60
0         60       409     Pulse        117       Pulse        103
1         60       479     Maxpulse     145       Maxpulse     135
2         60       340     Calories     479       Calories     340
3         45       282     Name: 1, dtype: int64  Name: 2, dtype: int64
4         45       406
```

Working with Pandas

Example 6: Read and plot CSV data file.

```
df = pd.read_csv('../data/AirPassengers.csv')
print(df.head())

print(df.info())  # <-- print dataframe info and shape ...
print(df.shape)
```

Output:

```
      Month  #Passengers
0   1949-01          112
1   1949-02          118
2   1949-03          132
3   1949-04          129
4   1949-05          121
```

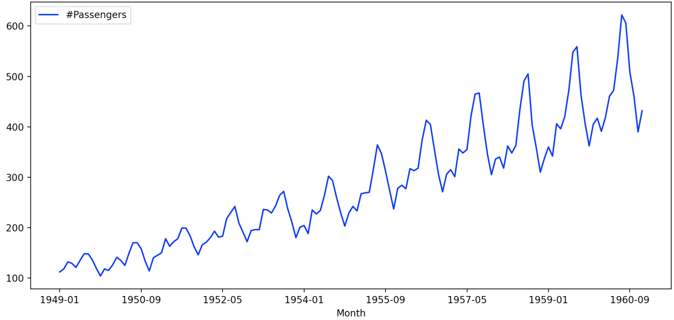
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
#   Column       Non-Null Count  Dtype
---  ------       --------------  -----
0   Month        144 non-null    object
1   #Passengers  144 non-null    int64
dtypes: int64(1), object(1)
memory usage: 2.4+ KB
None
(144, 2)
```

Working with Pandas

Example 6: (continued)

```
import matplotlib.pyplot as plt

ax = plt.gca()
df.plot(kind='line',x='Month',y='#Passengers',color='blue',ax=ax)
plt.show()
```

Output:



Spatial Data and Dataset Transformation

(Working with GeoPandas)

GeoPandas

GeoPandas

GeoPandas is an open source project to make working with geospatial data in Python easier.

Approach:

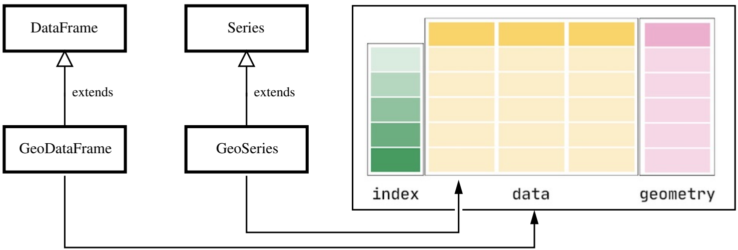
- Extend the datatypes used by Pandas to allow spatial operations on geometric types.
- Geometric operations are performed by shapely.
- Geopandas further depends on fiona for file access and matplotlib for plotting.

Installation

```
prompt >> pip3 install geopandas
```

Working with GeoPandas Dataframes

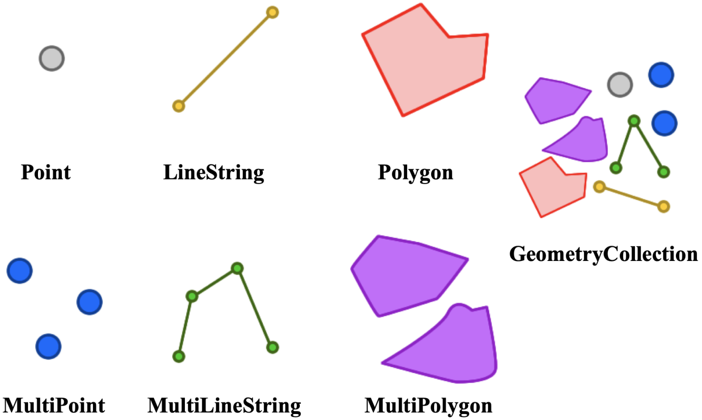
Core Modeling Concepts and Data Structure:



- GeoSeries handle geometries (points, polygons, etc).
- GeoDataFrames store geometry columns and perform spatial operations. They can be assembled from geopandas.GeoSeries.

Working with GeoPandas Dataframes

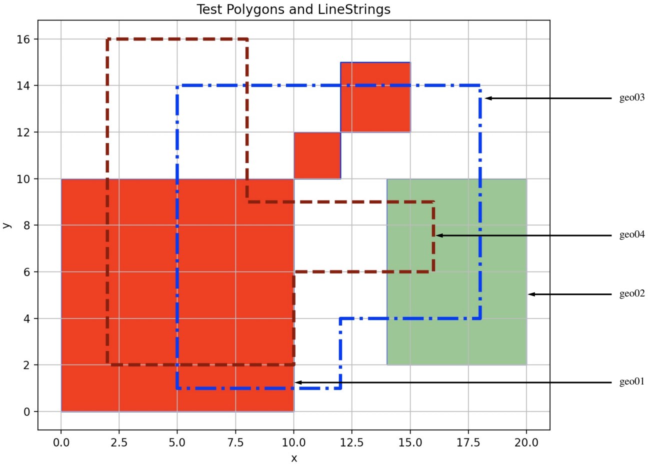
Geometric Objects: points, multi-points, lines, multi-lines, polygons, multi-polygons.



Example 1: Manual Specification of Geometric Shapes

Example 1: Manual specification of polygon and linestring shapes

...



Example 1: Manual Specification of Geometric Shapes

Part I: Problem Setup

```
# ===========================================================================
# TestGeoSeries01.py. Manual assembly of simple geometries.
#
# Written by: Mark Austin                                      February 2023
# ===========================================================================

import geopandas
from geopandas import GeoSeries
from shapely.geometry import Polygon
from shapely.geometry import LineString

import matplotlib.pyplot as plt

# ==============================
# main method ...
# ==============================

def main():
    print("--- Enter TestGeoSeries01.main()              ... ");
    print("--- ======================================== ... ");

    print("--- Part 01: Create individual polygons ... ");

    polygon01 = Polygon([   (0,0),  (10,0), (10,10),  (0,10) ] )
    polygon02 = Polygon([ (10,10), (12,10), (12,12), (10,12) ] )
    polygon03 = Polygon([ (12,12), (15,12), (15,15), (12,15) ] )
```

Example 1: Manual Specification of Geometric Shapes

Part I: Problem Setup (Continued) ....

```
    polygon04 = Polygon([ (14,2), (20,2), (20,10), (14,10) ] )

    print("--- Part 02: Add polygons to GeoSeries ... ");

    geo01 = GeoSeries( [ polygon01, polygon02, polygon03 ]);
    geo02 = GeoSeries( [ polygon04 ]);

    print("--- Part 03: Create simple linestring GeoSeries ... ");

    line01 = LineString([ (18,14), (5,14), (5,1), (12,1), (12,4), (18,4), (18,14) ] )
    geo03  = GeoSeries( [ line01 ]);
    line02 = LineString([ (2,16), (2,2), (10,2), (10,6), (16,6), (16,9), (8,9), (8,16),
    geo04  = GeoSeries( [ line02 ]);

    print("--- Part 04: Print GeoSeries info and contents ... ");

    print(geo01)
    print(geo02)

    print("--- Part 05: Area and boundary of geo01 ... ");

    print(geo01.area)
    print(geo01.boundary)

    print("--- Part 06: Area and boundary of geo02 ... ");

    print(geo02.area)
    print(geo02.boundary)
```

Example 1: Manual Specification of Geometric Shapes

Part I: Problem Setup (Continued) ....

```

print("--- Part 07: Spatial relationship of geo01 through geo04 ... ");

print("--- Compute intersection of (lines) geo03 and geo04 ...")
geo02a = geo03.intersects(geo04)
print("---   geo03.intersects(geo04) --> {:s} ...".format( str( geo02a[0] ) ))
geo02b = geo03.intersection(geo04)
print("---   geo03.intersection(geo04) --> {:s} ...".format( str( geo02b[0] ) ))

print("--- Compute intersection of (region) geo01 and (lines) geo03 and geo04 ...")
geo02c = geo01.intersection(geo03)
print("---   geo01.intersection(geo03) --> {:s} ...".format( str( geo02c[0] ) ))
geo02d = geo01.intersection(geo04)
print("---   geo01.intersection(geo04) --> {:s} ...".format( str( geo02d[0] ) ))

print("--- Compute intersection of (region) geo02 and (lines) geo03 and geo04 ...")
geo02e = geo02.intersection(geo03)
print("---   geo02.intersection(geo03) --> {:s} ...".format( str( geo02e[0] ) ))
geo02f = geo02.intersection(geo04)
print("---   geo02.intersection(geo04) --> {:s} ...".format( str( geo02f[0] ) ))

print("--- Part 08: Plot polygons ... ");

ax = geo01.plot( color='blue', edgecolor='black')
ax.set_aspect('equal')
ax.set_title("Test Polygons and LineStrings")
```

Example 1: Manual Specification of Geometric Shapes

Part I: Problem Setup (Continued) ....

```

    # Plot polygons ...

    geo01.plot(ax=ax, edgecolor='blue', color='red',   alpha= 1.0 )
    geo02.plot(ax=ax, edgecolor='blue', color='green', alpha= 0.5 )

    # Plot linestring ...

    geo03.plot(ax=ax, color='blue',   alpha= 1.0, linewidth=3.0, linestyle='dashdot' )
    geo04.plot(ax=ax, color='maroon', alpha= 1.0, linewidth=3.0, linestyle='dashed' )

    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid(True)
    plt.show()

    print("--- ======================================== ... ");
    print("--- Leave TestGeoSeries01.main()              ... ");

# ====================================================
# call the main method ...
# ====================================================

main()
```

Source Code: See: python-code.d/geopandas/

Example 1: Manual Specification of Geometric Shapes

Part II: Abbreviated Output:

```
--- Enter TestGeoSeries01.main()            ...
--- Part 01: Create individual polygons ...
--- Part 02: Add polygons to GeoSeries ...
--- Part 03: Create simple linestring GeoSeries ...

--- Part 04: Print GeoSeries info and contents ...

0    POLYGON ((0.00000 0.00000, 10.00000 0.00000, 1...
1    POLYGON ((10.00000 10.00000, 12.00000 10.00000...
2    POLYGON ((12.00000 12.00000, 15.00000 12.00000...
dtype: geometry
0    POLYGON ((14.00000 2.00000, 20.00000 2.00000, ...
dtype: geometry

--- Part 05: Area and boundary of geo01 ...

0    100.0
1      4.0
2      9.0
dtype: float64
0    LINESTRING (0.00000 0.00000, 10.00000 0.00000,...
1    LINESTRING (10.00000 10.00000, 12.00000 10.000...
2    LINESTRING (12.00000 12.00000, 15.00000 12.000...
dtype: geometry
```

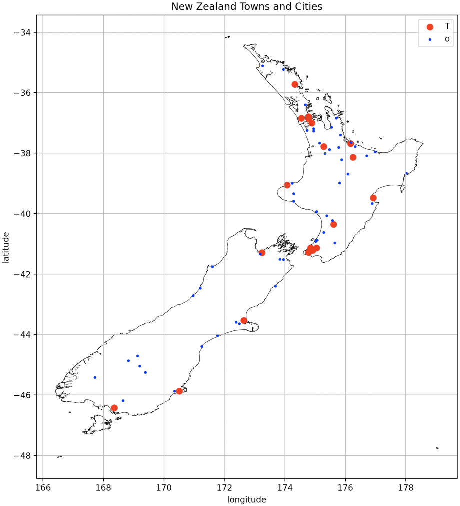
Example 1: Manual Specification of Geometric Shapes

Part II: Abbreviated Output:

```
--- Part 06: Area and boundary of geo02 ...

0    48.0
dtype: float64
0    LINESTRING (14.00000 2.00000, 20.00000 2.00000...
dtype: geometry

--- Part 07: Spatial relationship of geo01 through geo04 ...

--- Compute intersection of (lines) geo03 and geo04 ...

---   geo03.intersects(geo04) --> True ...
---   geo03.intersection(geo04) --> MULTIPOINT (5 2, 8 14) ...

--- Compute intersection of (region) geo01 and (lines) geo03 and geo04 ...

---   geo01.intersection(geo03) --> LINESTRING (5 10, 5 1, 10 1) ...
---   geo01.intersection(geo04) --> MULTILINESTRING ((10 2, 10 6), (2 10, 2 2, 10 2), (10 9, 8 9, 8 10))

--- Compute intersection of (region) geo02 and (lines) geo03 and geo04 ...

---   geo02.intersection(geo03) --> LINESTRING (14 4, 18 4, 18 10) ...
---   geo02.intersection(geo04) --> LINESTRING (14 6, 16 6, 16 9, 14 9) ...

--- Part 08: Plot polygons ...
--- Leave TestGeoSeries01.main()            ...
```

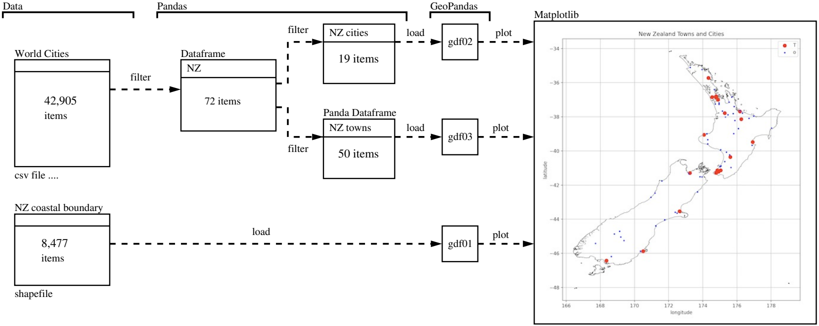
Example 2: Towns and Cities in New Zealand

Example 2: Towns and Cities in New Zealand.



Example 2: Towns and Cities in New Zealand

Part I: Data Processsing Pipeline: Use sequence of filters to specialize views of data ...



Example 2: Towns and Cities in New Zealand

Part II: Program Source Code:

```
# ========================================================================
# TestNewZealandDataModel.py. Assemble data model for towns and cities in
# New Zealand.
#
# Written by: Mark Austin                                    February 2023
# ========================================================================

from pandas import DataFrame
from pandas import Series
from pandas import read_csv

import numpy as np
import pandas as pd
import geopandas

import matplotlib.pyplot as plt

# =============================
# main method ...
# =============================

def main():
    print("--- Enter TestNewZealandDataModel.main()     ... ");
    print("--- ======================================== ... ");

    print("--- Part 01: Load world city dataset ... ");
```

Example 2: Towns and Cities in New Zealand

Part II: Program Source Code: (Continued) ...

```

    df = pd.read_csv("../data/cities/world-cities.csv")

    print("--- Part 02: Print dataframe info and contents ... ");

    print(df)
    print(df.info() )

    print("--- Part 03: Filter dataframe to keep only cities from New Zealand ... ")

    options = ['New Zealand']
    dfNZ        = df [ df['country'].isin(options) ].copy()

    print("--- Part 04: Filter data to find NZ cities and towns ... ")

    dfNZcities = dfNZ [ (dfNZ['population'] > 40000) ].sort_values( by=['population'] )

    dfNZtowns  = dfNZ [ (dfNZ['population'] > 1000) & (dfNZ['population'] < 40000) ]
    dfNZtowns  = dfNZtowns.sort_values( by=['population'] )

    print('--- New Zealand Cities:\n', dfNZcities )
    print('--- New Zealand Towns:\n', dfNZtowns )

    print("--- Part 05: Read NZ coastline shp file into geopandas ... ")

    nzboundarydata = geopandas.read_file("../data/geography/nz/Coastline02.shp")
    print(nzboundarydata)
```

Example 2: Towns and Cities in New Zealand

Part II: Program Source Code: (Continued) ...

```
    print("--- Part 06: Define geopandas dataframes ... ")

    gdf01 = geopandas.GeoDataFrame(nzboundarydata)
    gdf02 = geopandas.GeoDataFrame( dfNZcities,
                 geometry=geopandas.points_from_xy(dfNZcities.lng, dfNZcities.lat))
    gdf03 = geopandas.GeoDataFrame(  dfNZtowns,
                 geometry=geopandas.points_from_xy( dfNZtowns.lng, dfNZtowns.lat))

    print(gdf01.head())

    print("--- Part 07: Create boundary map for New Zealand ... ")

    # We can now plot our ``GeoDataFrame``.

    ax = gdf01.plot( color='white', edgecolor='black')
    ax.set_aspect('equal')
    ax.set_title("New Zealand Towns and Cities")

    gdf01.plot(ax=ax, color='white')

    gdf02.plot(ax=ax, color =  'red', markersize = 50, label= 'Cities')
    gdf03.plot(ax=ax, color = 'blue', markersize =  5, label= 'Towns' )

    plt.legend('Towns/Cities:')
    plt.xlabel('longitude')
    plt.ylabel('latitude')
```

Example 2: Towns and Cities in New Zealand

Part II: Program Source Code: (Continued) ...

```
    plt.grid(True)
    plt.show()

    print("--- ======================================== ... ");
    print("--- Leave TestNewZealandDataModel.main()     ... ");

# ==================================================
# call the main method ...
# ==================================================

main()
```

Source Code: See: python-code.d/geopandas/

Example 2: Towns and Cities in New Zealand

Part III: Abbreviated Output:

```
--- Enter TestNewZealandDataModel.main()     ...
--- ======================================== ...
--- Part 01: Load world city dataset ...
--- Part 02: Print dataframe info and contents ...
             city  city_ascii      lat  ... capital population         id
0           Tokyo       Tokyo  35.6839  ... primary 39105000.0 1392685764
1         Jakarta     Jakarta  -6.2146  ... primary 35362000.0 1360771077
...           ...         ...      ...  ...     ...        ...        ...
42903 Timmiarmiut Timmiarmiut  62.5333  ...     NaN       10.0 1304206491
42904     Nordvik     Nordvik  74.0165  ...     NaN        0.0 1643587468
[42905 rows x 11 columns]

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 42905 entries, 0 to 42904
Data columns (total 11 columns):
 #   Column      Dtype            #   Column      Dtype
---  ------      -----           ---  ------      -----
 0   city        object           6   iso3        object
 1   city_ascii  object           7   admin_name  object
 2   lat         float64          8   capital     object
 3   lng         float64          9   population  float64
 4   country     object          10   id          int64
 5   iso2        object
dtypes: float64(3), int64(1), object(7)
memory usage: 3.6+ MB
```

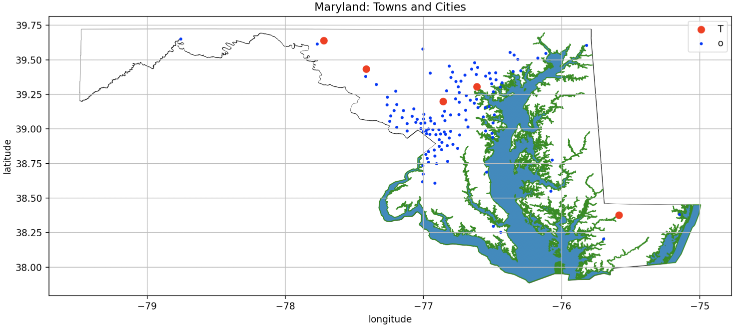
Example 2: Towns and Cities in New Zealand

Part III: Abbreviated Output (Continued) ...

```
--- Part 03: Filter dataframe to keep only cities from New Zealand ...
--- Part 04: Filter data to find NZ cities and towns ...

--- New Zealand Cities:
                  city        city_ascii  ...  population          id
14169        Upper Hutt       Upper Hutt  ...     41000.0  1554000042
6159       Invercargill     Invercargill  ...     47625.0  1554148942
.....
741          Wellington       Wellington  ...    418500.0  1554772152
516            Auckland         Auckland  ...   1346091.0  1554435911
[19 rows x 11 columns]

--- New Zealand Towns:
                 city       city_ascii  ...  population          id
42142         Kaikoura        Kaikoura  ...      2210.0  1554578431
.....
14309        Whanganui       Whanganui  ...     39400.0  1554827998
[50 rows x 11 columns]

--- Part 05: Read NZ coastline shp file into geopandas ...
0     POLYGON ((174.00369 -40.66489, 174.00372 -40.6...
.....
8476  POLYGON ((173.01384 -34.39348, 173.01395 -34.3...
[8477 rows x 1 columns]
--- Part 07: Create boundary map for New Zealand ...
--- ====================================== ...
--- Leave TestNewZealandDataModel.main()     ...
```

Example 3: Towns and Cities in Maryland

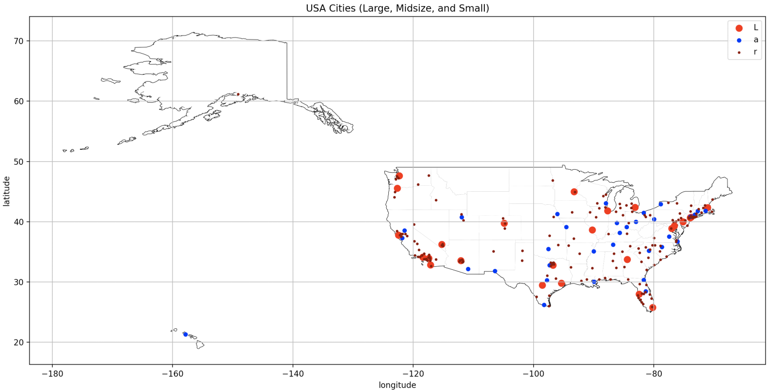
Example 3: Towns and Cities in Maryland.



Cities: Columbia (pop. 103991), Salisbury (pop. 106447), Frederick (pop. 156787), Hagerstown (pop. 184755), Baltimore (pop. 2106068).

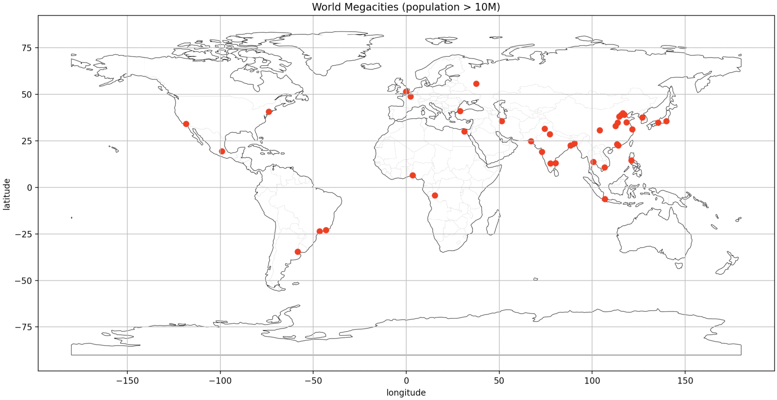
Example 4: Large, Midsize, and Small US Cities

Example 4: Large, Midsize, and Small US Cities



Cities: 26 large (pop. > 2M), 34 midsize (800k < pop. < 2M), 172 small (200k < pop. < 800k).

Example 5: The World's Megacities



Example 5: The World's Megacities

```
--- Part 02: Filter to keep only large cities (pop. > 10M) ...

                 city        city_ascii  ...  population          id
0               Tokyo             Tokyo  ...  39105000.0  1392685764
1             Jakarta           Jakarta  ...  35362000.0  1360771077
2               Delhi             Delhi  ...  31870000.0  1356872604
3              Manila            Manila  ...  23971000.0  1608618140
4           São Paulo         Sao Paulo  ...  22495000.0  1076532519
5               Seoul             Seoul  ...  22394000.0  1410836482
6              Mumbai            Mumbai  ...  22186000.0  1356226629
7            Shanghai          Shanghai  ...  22118000.0  1156073548
8         Mexico City       Mexico City  ...  21505000.0  1484247881
9           Guangzhou         Guangzhou  ...  21489000.0  1156237133
10              Cairo             Cairo  ...  19787000.0  1818253931
11            Beijing           Beijing  ...  19437000.0  1156228865
12           New York          New York  ...  18713220.0  1840034016
13            Kolkāta           Kolkata  ...  18698000.0  1356060520
14             Moscow            Moscow  ...  17693000.0  1643318494
15            Bangkok           Bangkok  ...  17573000.0  1764068610

... details removed ...

33             London            London  ...  11120000.0  1826645935
34              Paris             Paris  ...  11027000.0  1250015082
35            Tianjin           Tianjin  ...  10932000.0  1156174046
36              Linyi             Linyi  ...  10820000.0  1156086320
37       Shijiazhuang      Shijiazhuang  ...  10784600.0  1156217541
38          Zhengzhou         Zhengzhou  ...  10136000.0  1156183137
39            Nanyang           Nanyang  ...  10013600.0  1156192287
```