



# Overview

- 1 What is Python?
  - Origins, Features, Framework for Scientific Computing
- 2 Program Development with Python
  - Working with the Terminal
  - Integrated Development Environments
- 3 Elements of Python Programming
  - Data Types, Variables, Arithmetic Expressions, Strings, Program Control, and Functions
- 4 First Program (Evaluate and Plot Sigmoid Function)
- 5 Builtin Collections (Lists, Dictionaries, and Sets)
- 6 Numerical Python (NumPy)







# 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 !!!!













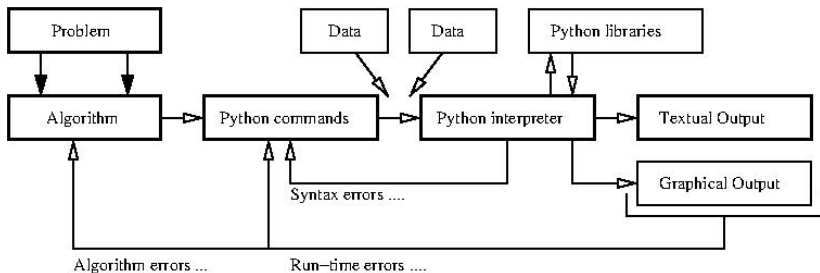






# First Steps: Working with the Terminal

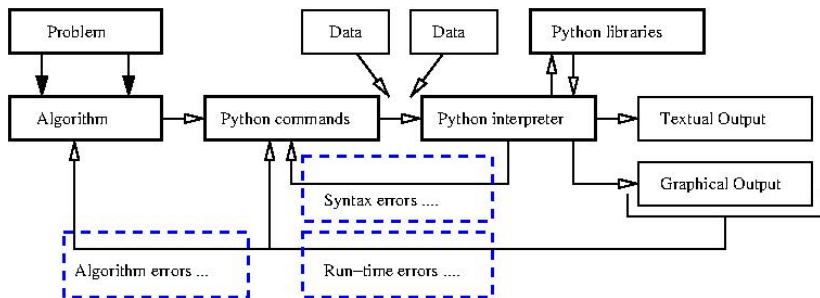
Program Development in the Terminal Window:



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?







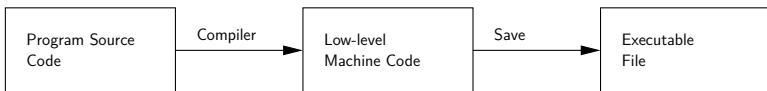
# Interpreted and Compiled Programming Languages

## 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:

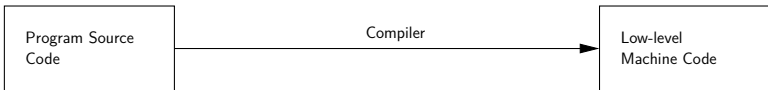
- Interpreted programs can modify themselves by adding or changing functions at runtime.
- Cycles of **application development** are **usually faster** than with compiled code because you don't have to recompile your application each time you want to test a small section.

# Interpreted and Compiled Programming Languages

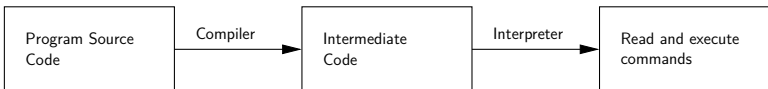
## Modern Interpreter Systems

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

### Compiled Code



### Compiled and Interpreted Code



Examples: Java and Python (even MATLAB).



# 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

The screenshot displays the Visual Studio Code interface. The Explorer sidebar on the left shows a project structure with folders for 'HELLO' and 'standardplot.py'. The main editor window displays the code for 'standardplot.py':

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 20, 100) # Create a list of evenly-spaced numbers
5 plt.plot(x, np.sin(x))      # Plot the sine of each x point
6 plt.show()
```

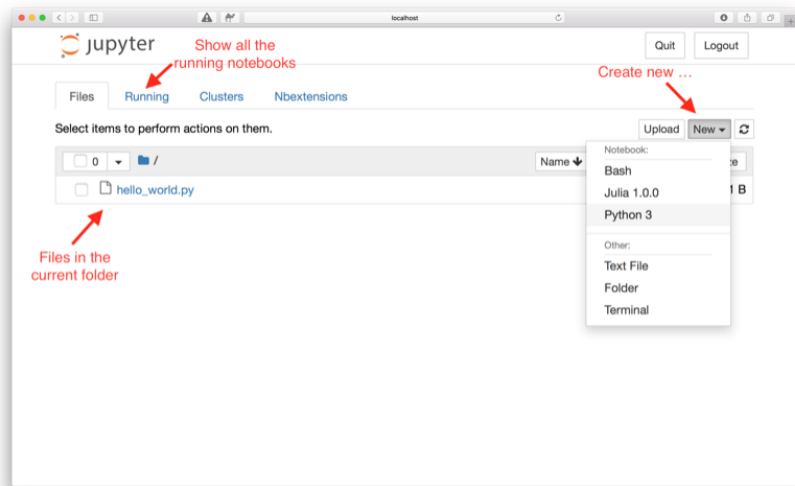
The 'Figure 1' window shows a plot of a sine wave. The x-axis ranges from 0.0 to 20.0 with major ticks every 2.5 units. The y-axis ranges from -1.00 to 1.00 with major ticks every 0.25 units. The plot shows three full cycles of a sine wave, starting at (0,0), peaking at approximately (1.57, 1.0), crossing the x-axis at approximately (3.14, 0), reaching a trough at approximately (4.71, -1.0), and completing a cycle at approximately (6.28, 0). The plot is titled 'Figure 1' and has standard window controls.

The bottom status bar shows the Python 3.9.6 64-bit (.venv) environment, the current cursor position (Ln 6, Col 14), and the file encoding (UTF-8, CRLF).





# Jupyter Notebook User Interface



# Jupyter Notebook User Interface

The screenshot shows the Jupyter Notebook interface in a web browser. The browser address bar shows `localhost:8891/notebooks/hello_world.ipynb`. The notebook title is `hello_world` and it indicates `Last Checkpoint: a minute ago (unsaved changes)`. The interface includes a menu bar with `File`, `Edit`, `View`, `Insert`, `Cell`, `Kernel`, `Widgets`, and `Help`. A toolbar below the menu bar contains icons for `Undo`, `Redo`, `Run`, `Stop`, `Clear`, `Insert`, `Markdown`, and `Code`. The main content area is divided into two cells. The top cell is a code cell containing `print('Hello World')` and has output `Hello World`. The bottom cell is a markdown cell containing `# This is a markdown cell (header level 1)`, `## Header level 2`, `You can use bold text`, `You can use bullets list:`, and a bulleted list. Red arrows point to various UI elements: the header, menu, toolbar, code cell, code cell outputs, raw markdown cell, and rendered markdown cell.

Annotations in the image:

- Header
- Menu
- Toolbar
- Code cell, press Shift + Enter to run
- Code cell outputs
- Raw Markdown cell after double click
- Rendered Markdown cell after pressing Shift + Enter

# 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 notation.
- **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)

The screenshot displays a Jupyter Notebook window titled "01\_the\_machine\_learning\_landscape". The interface includes a top menu bar with options like File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with icons for file operations and execution. The main content area shows the following text:

**Chapter 1 – The Machine Learning landscape**

*This is the code used to generate some of the figures in chapter 1.*

[Run in Google Colab](#)

**Code example 1-1**

Although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead.

```
In [1]: # Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)
```

```
In [2]: # Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"
```

This function just merges the OECD's life satisfaction data and the IMF's GDP per capita data. It's a bit too long and boring and it's not specific to Machine Learning, which is why I left it out of the book.

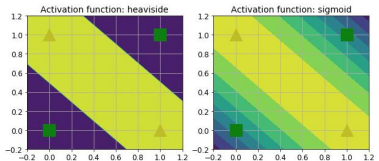
On the right side of the notebook, there is a book cover for "Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow" by Aurélien Géron, published by O'Reilly. The cover features a yellow and black salamander.

# Jupyter Notebook and Machine Learning

## Jupyter Notebook (Machine Learning with TensorFlow)

```
In [7]: 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)
```

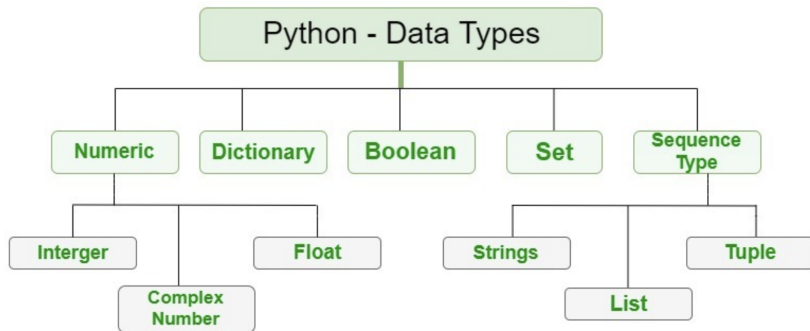
```
In [8]: x1s = np.linspace(-0.2, 1.2, 100)  
        x2s = np.linspace(-0.2, 1.2, 100)  
        x1, x2 = np.meshgrid(x1s, x2s)  
  
        z1 = mlp_xor(x1, x2, activation=heaviside)  
        z2 = mlp_xor(x1, x2, activation=sigmoid)  
  
        plt.figure(figsize=(10,4))  
  
        plt.subplot(121)  
        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)
```





# 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 builtin 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']
```

# 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 permutations  $\rightarrow$  a working range  $[-2.147, 2.147]$  billion.

# 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:

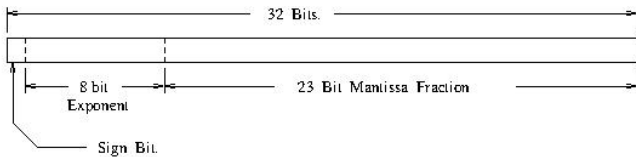
$$X = \sigma \cdot m \cdot 2^E. \quad (1)$$

Here:

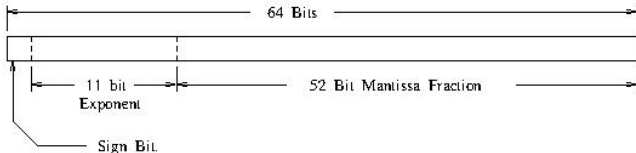
- $\sigma$  represents the sign of the number.
- $m$  is the mantissa (interpreted as a fraction  $0 < m < 1$ ).
- $E$  is the exponent.

# IEEE 754 Floating-Point Standard

Ensures floating point implementations 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

## Accessing the Math Library Module

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

## Math Constants

Method	Description
<code>math.e</code>	Returns Euler's number (2.7182 ...).
<code>math.inf</code>	Returns floating-point positive infinity.
<code>math.pi</code>	Returns PI (3.1415926 ...).

## Math Methods

Method	Description
<code>math.acos()</code>	Returns the arc cosine of a number.
<code>math.acosh()</code>	Returns the inverse hyperbolic cosine of a number.
<code>math.asin()</code>	Returns the arc sine of a number.
<code>math.asinh()</code>	Returns the inverse hyperbolic sine of a number.

# Working with Double Precision Numbers

## Math Methods (continued) ...

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

# Working with Double Precision Numbers

## Math Methods (continued) ...

Method	Description
<code>math.log()</code>	Returns the natural logarithm of a number, or the logarithm of number to base.
<code>math.log10()</code>	Returns the base-10 logarithm of x
<code>math.log1p()</code>	Returns the natural logarithm of 1+x
<code>math.log2()</code>	Returns the base-2 logarithm of x
<code>math.perm()</code>	Returns the number of ways to choose k items from n items with order and without repetition
<code>math.pow()</code>	Returns the value of x to the power of y
<code>math.prod()</code>	Returns the product of all the elements in an iterable
<code>math.radians()</code>	Converts a degree value into radians
<code>math.remainder()</code>	Returns the closest value that can make numerator completely divisible by the denominator
<code>math.sin()</code>	Returns the sine of a number
<code>math.sinh()</code>	Returns the hyperbolic sine of a number
<code>math.sqrt()</code>	Returns the square root of a number
<code>math.tan()</code>	Returns the tangent of a number
<code>math.tanh()</code>	Returns the hyperbolic tangent of a number
<code>math.trunc()</code>	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 power of "b".	$2**3 = 2*2*2 = 8$
*	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 operations are evaluated left to right.
+ -	4	Addition and subtraction operations are evaluated left to right.

```
=====
```



# 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-hand
side multiplication.
==> 4*math.sin( math.pi/4 + math.pi/4 ) <== evaluate expression within
function parentheses, start
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

$$\sin(x)^2 + \cos(x)^2 = 1.0 \quad (2)$$

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 multiplication 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 ...
```

# Strings

# Strings

## String

A string is sequence of characters (letters, numbers, punctuation, spaces, etc) enclosed in quotes.

### Three ways to create a string:

```
a = '20'           # <--- string with single quotes ...
b = "Hello World" # <--- string with double quotes ...
c = """This is a
    multiline
    string."""
d = "dogs"
e = "cats"
```

**Note:** Strings are immutable – once created they cannot be changed.

# Strings

## Traditional style of String formatting:

```
print("--- String a: {:s} ...".format(a) );
print("--- String b: {:s} ...".format(b) );
print("--- String c: {:s} ...".format(c) );
print("--- Multiple substitutions: {:s} chasing {:s} ...".format( d, e ) );
```

## Using f-String (formatted string literals):

```
print(f"--- String: b = {b} ...");
print(f"--- Multiple substitutions: {d} chasing {e} ..." );
```

# Strings

**Accessing individual string characters:** The first character is at index 0.

```
print("--- Indexing b[0]: {:s} ...".format( b[0] ) );
print("--- Indexing b[1]: {:s} ...".format( b[1] ) );
print("--- Indexing b[2]: {:s} ...".format( b[2] ) );
print("--- Indexing b[3]: {:s} ...".format( b[3] ) );
print("--- Indexing b[4]: {:s} ...".format( b[4] ) );
```

## Useful string methods:

```
print("--- Uppercase string b: {:s} ...".format( b.upper() ) );
print("--- Lowercase string b: {:s} ...".format( b.lower() ) );
```

```
# Replace "World" with "ENCE 201"
```

```
print("--- Modified string b: {:s} ...".format( b.replace("World","ENCE 201") ) )
```

**Source Code:** See [python-code.d/basics/TestStrings.py](https://python-code.d/basics/TestStrings.py)



# Program Control

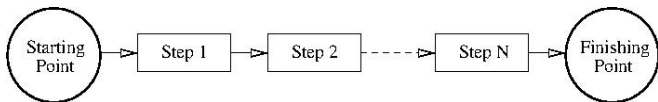
Behavior models coordinate a set of what we will call steps. Two questions need to be answered at each step:

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

Abstractions that allow for the ordering of functions include:

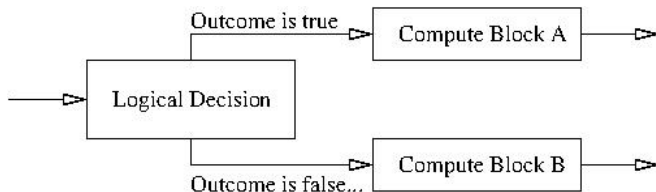
- Sequence constructs,
- Branching constructs,
- Repetition/looping constructs,

## Sequences:

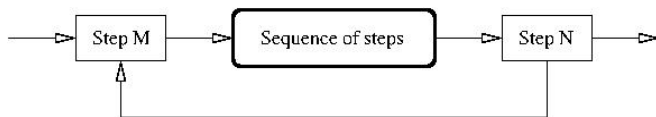


# Program Control Abstractions

## 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

## 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

```
=====
i = 1
while i <= 10:
    print("--- i = {:.2f} ...".format(i) )
    i = i + 2
```

Program Output

```
=====
--- i = 1.00 ...
--- i = 3.00 ...
--- i = 5.00 ...
--- i = 7.00 ...
--- i = 9.00 ...
```

## Example 2: Simple while loop with break statement.

Python Code

```
=====
i = 1
while i <= 10:
    print("--- i = {:.2f} ...".format(i) )
    if i == 5:
        break
    i = i + 2
```

Program Output

```
=====
--- i = 1.00 ...
--- i = 3.00 ...
--- i = 5.00 ...
```

# Looping Constructs

## Example 3: Simple while loop with continue statement.

Python Code

```

=====
i = 1
while i <= 10:
    print("--- i = {:.5.2f} ...".format(i) )
    if i == 5:
        i = i + 1
        continue
    i = i + 2

```

Program Output

```

=====
--- i = 1.00 ...
--- i = 3.00 ...
--- i = 5.00 ...
--- i = 6.00 ...
--- i = 8.00 ...
--- i = 10.00 ...

```

## Example 4: While loop with else condition ...

Python Code

```

=====
i = 1
while i < 6:
    print("--- i = {:.2f} ...".format(i) )
    i += 1
else:
    print("--- i no longer less than 6")

```

Program Output

```

=====
--- i = 1.00 ...
--- i = 2.00 ...
--- i = 3.00 ...
--- i = 4.00 ...
--- i = 5.00 ...
--- i no longer less than 6

```



# 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: Strategies for Handling Complexity

## Function

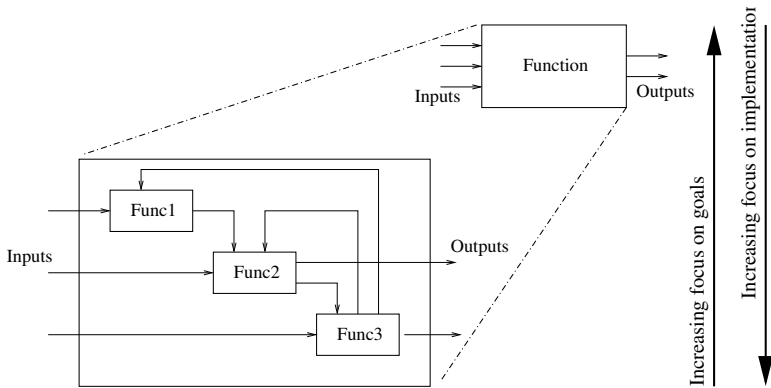
A **function** is a **block of reusable code** that performs a specific task. Instead of writing the same code over and over again, we define a function and call it when needed.

### Use Cases:

- Avoid repeating code ...
- Helps to organize code into modules.
- Simplifies identification of bugs and software maintenance.

# Functions: Strategies for Handling Complexity

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



# Functions: Syntax and Types

## Syntax:

```
def function-name ( parameters ):  
    # block of code ...  
  
    return result
```

## Types of Function:

- Builtin functions ...
- User-defined functions ...
- Lambda functions ...

# Python: Builtin Functions

Built-in Functions			
<b>A</b> abs() aiter() all() any() anext() ascii()	<b>E</b> enumerate() eval() exec()	<b>L</b> len() list() locals()	<b>R</b> range() repr() reversed() round()
<b>B</b> bin() bool() breakpoint() bytearray() bytes()	<b>F</b> filter() float() format() frozenset()	<b>M</b> map() max() memoryview() min()	<b>S</b> set() setattr() slice() sorted() staticmethod() str() sum() super()
<b>C</b> callable() chr() classmethod() compile() complex()	<b>G</b> getattr() globals()	<b>N</b> next()	<b>T</b> tuple() type()
<b>D</b> delattr() dict() dir() divmod()	<b>H</b> hasattr() hash() help() hex()	<b>O</b> object() oct() open() ord()	<b>V</b> vars()
	<b>I</b> id() input() int() isinstance() issubclass() iter()	<b>P</b> pow() print() property()	<b>Z</b> 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 Description

## 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:

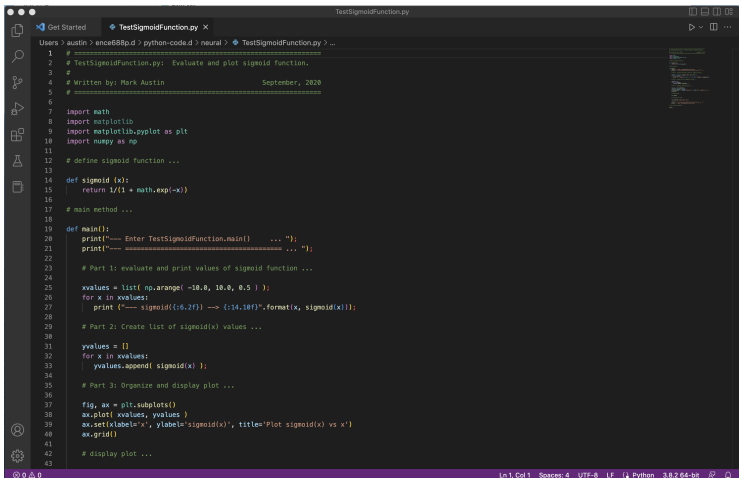
```
prompt >> pip3 list
```

# 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



```
1 # =====
2 # TestSigmoidFunction.py: Evaluate and plot sigmoid function.
3 #
4 # Written by: Mark Austin          September, 2020
5 # =====
6
7 import math
8 import matplotlib
9 import matplotlib.pyplot as plt
10 import numpy as np
11
12 # define sigmoid function ...
13
14 def sigmoid (x):
15     return 1/(1 + math.exp(-x))
16
17 # main method ...
18
19 def main():
20     print("--- Enter TestSigmoidFunction.main() ---");
21     print("--- ===== ... ---");
22
23     # Part 1: evaluate and print values of sigmoid function ...
24
25     xvalues = list( np.arange( -10.0, 10.0, 0.5 ) );
26     for x in xvalues:
27         print ( f"--- sigmoid({:6.2f}) --> {:-14.10f}".format(x, sigmoid(x)) );
28
29     # Part 2: Create list of sigmoid(x) values ...
30
31     yvalues = []
32     for x in xvalues:
33         yvalues.append( sigmoid(x) );
34
35     # Part 3: Organize and display plot ...
36
37     fig, ax = plt.subplots()
38     ax.plot( xvalues, yvalues )
39     ax.set(xlabel='x', ylabel='sigmoid(x)', title='Plot sigmoid(x) vs x')
40     ax.grid()
41
42     # display plot ...
43
```

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



# Program Source Code

```
1  # =====
2  # TestSigmoidFunction.py: Evaluate/plot sigmoid function.
3  #
4  # Written by: Mark Austin           September, 2020
5  # =====
6
7  import math
8  import matplotlib
9  import matplotlib.pyplot as plt
10 import numpy as np
11
12 # define sigmoid function ...
13
14 def sigmoid (x):
15     return 1/(1 + math.exp(-x))
16
17 # main method ...
18
19 def main():
20     print("--- Enter TestSigmoidFunction.main() ...");
21     print("--- =====");
22
23     # Part 1: Evaluate and print sigmoid function
24
25     xvalues = list( np.arange( -10.0, 10.0, 0.5 ) );
26     for x in xvalues:
27         print ("--- sigmoid({:6.2f}) --> {:14.10f}".format(x, sigmoid(x)));
28
29     # Part 2: Create list of sigmoid(x) values ...
```

# Program Source Code

```
29     # Part 2: Create list of sigmoid(x) values ...
30
31     yvalues = []
32     for x in xvalues:
33         yvalues.append( sigmoid(x) );
34
35     # Part 3: Organize and display plot ...
36
37     fig, ax = plt.subplots()
38     ax.plot( xvalues, yvalues )
39     ax.set(xlabel='x', ylabel='sigmoid(x)',
40           title='Plot sigmoid(x) vs x')
41     ax.grid()
42
43     # display and save plot ...
44
45     plt.show()
46
47     fig.savefig("sigmoid-plot.jpg")
48
49     print("--- ===== ...");
50     print("--- Leave TestSigmoidFunction.main() ...");
51
52     # call the main method ...
53
54     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.

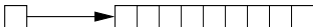
# Working with Lists

## List

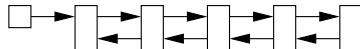
Lists are used to store multiple items in a single variable. A list may store multiple types (heterogeneous) of elements.

### Array, List, HashMap, and Queue Structures

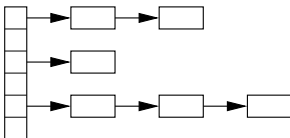
Arrays



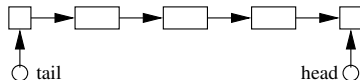
Linked List



Hash Map



Queues



# 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} ...".format( str( list01 ) ))
print ("--- List02: {:s} ...".format( str( list02 ) ))
```

# Create list with mix of data types ...

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

print ("--- List03 (with multiple data types): {:s} ...".format( str(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} ...".format( str( list04[0] ) ) )
print ( "--- list04[1]: {:s} ...".format( str( list04[1] ) ) )
print ( "--- list04[2]: {:s} ...".format( str( list04[2] ) ) )
print ( "--- list04[3]: {:s} ...".format( str( list04[3] ) ) )
print ( "--- list04[4]: {:s} ...".format( str( 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/](http://python-code.d/collections/)

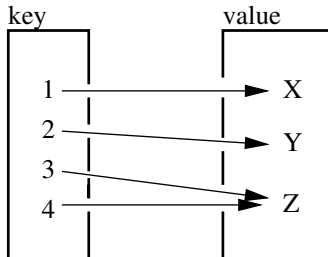
# Working with Dictionaries

## 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

### Maps



# Working with Dictionaries

## Basic Dictionary Methods

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

print ("--- Car01: {:s} ...".format( str( car01 ) ))
```

**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} ...".format( str( car01.get("brand")) ))
print ("---      : model --> {:s} ...".format( str( car01.get("model")) ))
print ("---      : miles --> {:d} ...".format( car01.get("miles") ))
print ("---      : new   --> {:s} ...".format( str ( car01.get("new")) ))
print ("---      : year  --> {:d} ...".format( 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/](http://python-code.d/collections/)

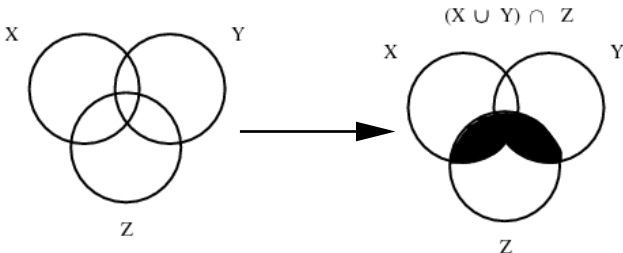
# Working with Sets

## 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

Sets



# 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/](http://python-code.d/collections/)



# 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 NumPy

<b>dtype</b>	<b>Variants</b>	<b>Description</b>
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 character 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.   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 $\rightarrow$ 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

```
0.000  1.000
3.000  4.000
```

Matrix: np.hstack(d1, d2)

```
0.000  1.000  5.000  6.000
3.000  4.000  7.000  8.000
```

Matrix: d2

```
5.000  6.000
7.000  8.000
```

Matrix: np.vstack(d1, d2)

```
0.000  1.000
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) identity 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

Matrix: matrix06

```
0.326
0.459
0.545
.....
0.803
0.014
0.291
```

--- Reshape to (10x2) matrix ...

Matrix: matrix06 (reshaped)

```
0.326  0.459
0.545  0.419
0.537  0.632
.....  .....
.....  .....
0.165  0.803
0.014  0.291
```

# Working with NumPy

## Example 10: Generate and plot linear space of coordinates:

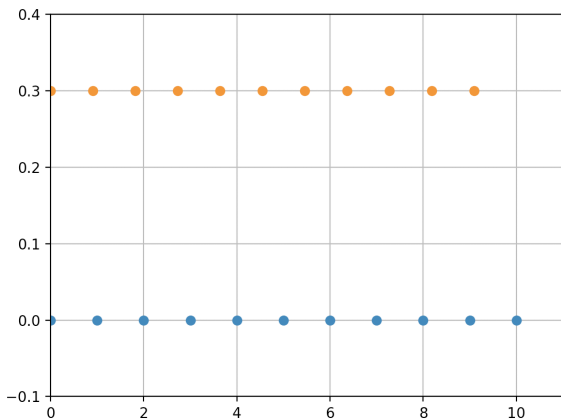
```

1  # =====
2  # TestLinspace01.py: Generate arrays of coordinates with np.linspace(), then plot.
3  # =====
4
5  import numpy as np # Make numpy available using np.
6  import matplotlib.pyplot as plt
7
8  def main():
9      # Generate arrays of x coordinates with np.linspace() ...
10
11     Npoints = 11
12     x1 = np.linspace(0, 10, num = Npoints, endpoint=True);
13     x2 = np.linspace(0, 10, num = Npoints, endpoint=False);
14
15     # Plot coordinates ...
16
17     y = np.zeros(Npoints)
18     plt.plot(x1, y, 'o')
19     plt.plot(x2, y + 0.3, 'o')
20     plt.ylim( [-0.1, 0.4] )
21     plt.xlim( [ 0.0, 11] )
22     plt.grid(); plt.show()
23
24 # call the main method ...
25
26 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:

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

# Working with NumPy

## Part II: Program Source Code:

```
1 # =====
2 # TestMatrixEquations01.py: Compute solution to matrix equations.
3 #
4 # Written by: Mark Austin                                November 2022
5 # =====
6
7 import numpy as np
8 from numpy.linalg import matrix_rank
9
10 # Function to print two-dimensional matrices ...
11
12 def PrintMatrix(name, a):
13     print("Matrix: {:s} ".format(name) );
14     for row in a:
15         for col in row:
16             print("{:8.3f}".format(col), end=" ")
17         print("")
18
19 # main method ...
20
21 def main():
22     print("--- Enter TestMatrixEquations01.main()      ... ");
23     print("--- ===== ... ");
24
25     print("--- Part 1: Create test matrices ... ");
```

# Working with NumPy

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

```
27     A = np.array( [ [ 3, -6,  7],
28                   [ 9,  0, -5],
29                   [ 5, -8,  6] ] );
30     PrintMatrix("A", A);
31
32     B = np.array([ [3], [3], [-4] ]);
33     PrintMatrix("B", B);
34
35     print("--- Part 2: Check properties of matrix A ... ");
36
37     rank = matrix_rank(A)
38     det  = np.linalg.det(A)
39
40     print("--- Matrix A: rank = {:f}, det = {:f} ...".format(rank, det) );
41
42     print("--- Part 3: Solve A.x = B ... ");
43
44     x = np.linalg.solve(A, B)
45     PrintMatrix("x", x);
46
47     print("--- ===== ... ");
48     print("--- Leave TestMatrixEquations01.main()      ... ");
49
50     # call the main method ...
51
52     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  $A \cdot x = B$  ...
```

```
Matrix: x
```

```
2.000
4.000
3.000
```