Python Tutorial – Part I: Introduction

Mark A. Austin

University of Maryland

austin@umd.edu

ENCE 201, Fall Semester 2023

September 3, 2023

Overview

1. What is Python?
   - Origins, Features, Framework for Scientific Computing
2. Program Development with Python
   - Working with the Terminal
   - Integrated Development Environments
3. Data Types, Variables, Arithmetic Expressions, Program Control, and Functions
4. First Program (Evaluate and Plot Sigmoid Function)
5. Builtin Collections (Lists, Dictionaries, and Sets)
6. Numerical Python (NumPy)
7. Tabular Data and Dataset Transformation (Pandas)
8. Spatial Data and Dataset Transformation (GeoPandas)

Part 2

Data Types

(Data Types in Python)

Integers

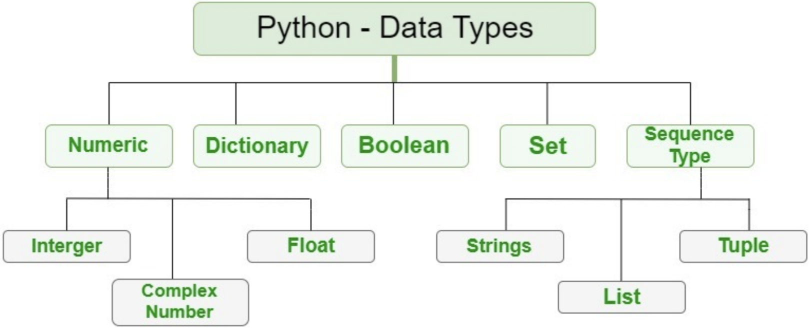
Requirements for storing 4 types of integer:

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:

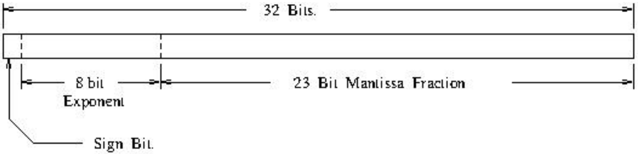
$$X = \sigma \cdot m \cdot 2^E. \tag{1}$$

Here:

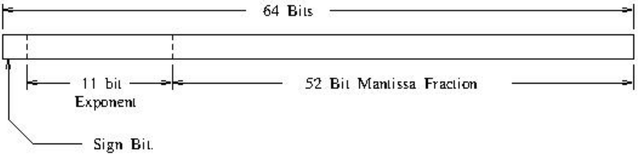
- σ represents the sign of the number.
- m is the mantissa (interpreted as a fraction $0 < m < 1$).
- E is the exponent.

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.



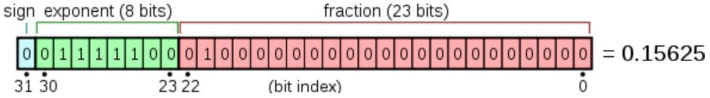
IEEE FLOATING POINT ARITHMETIC STANDARD FOR DOUBLE PRECISION FLOATS.

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   +- 11.7976931348623157OE+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

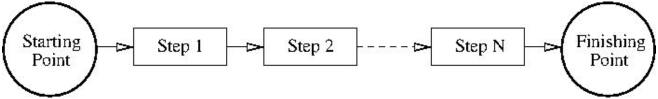
Behavior models coordinate a set of what we will call steps. Two questions need to be answered at each step:

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

Abstractions that allow for the ordering of functions include:

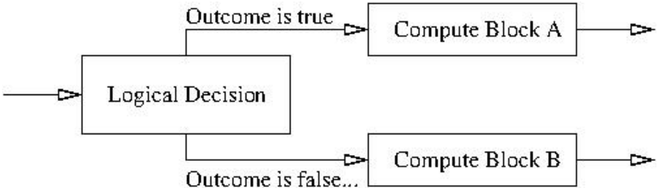
- Sequence constructs,
- Branching constructs,
- Repetition/looping constructs,

Sequences:

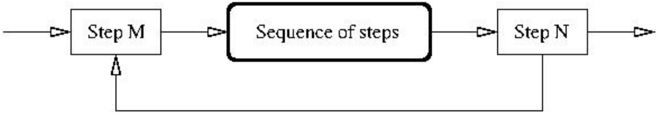


Program Control Abstractions

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:
   statement 2;
elif:
   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) )
```

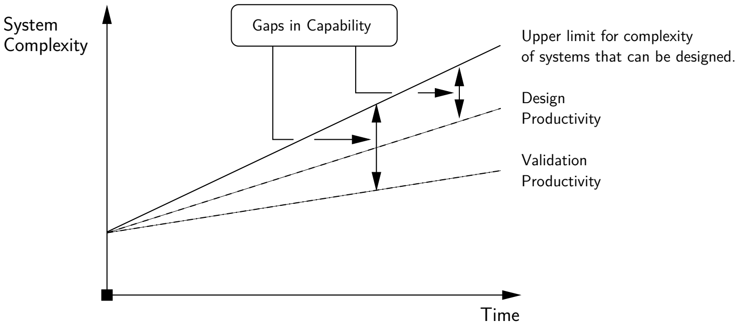
```
Program Output
=====================
--- red apple ...
--- red banana ...
--- red cherry ...
--- big apple ...
--- big banana ...
--- big cherry ...
--- tasty apple ...
--- tasty banana ...
--- tasty cherry ...
--- spoiled apple ...
--- spoiled banana ...
--- spoiled cherry ...
```

Functions

Functions: Strategies for Handling Complexity

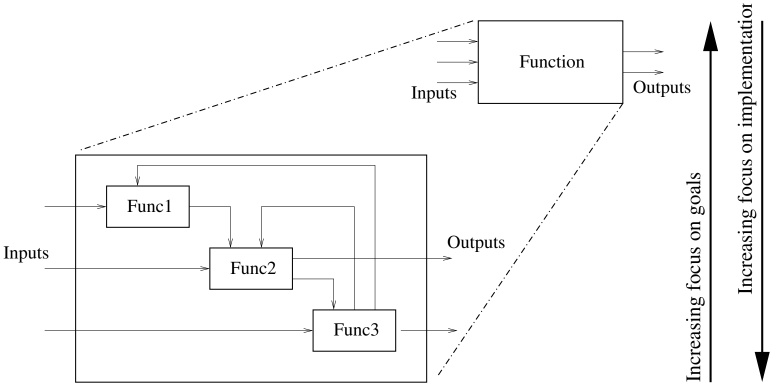
Productivity Concerns

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

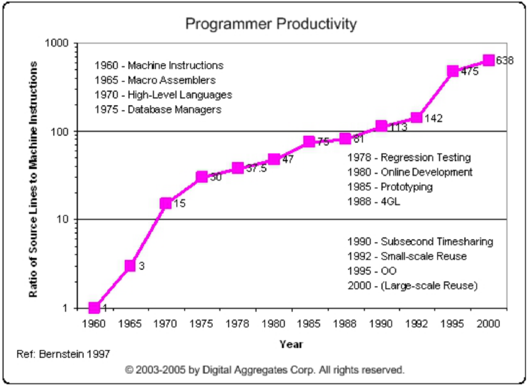
Simplify models of funtionality by decomposing high-level functions into networks of lower-level functionality:

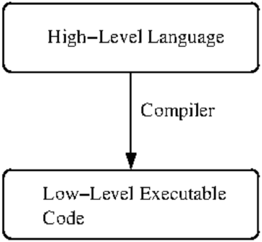


Functions: Strategies for Handling Complexity

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 ...
```