Python memento

TI-Smart Grids

Genoveva Vargas-Solar
French Council of Scientific Research, LIG
genoveva.vargas@imag.fr

http://vargas-solar.com/data-centric-smart-everything/

Thanks to Prof. J.L. Zechinelli Martini, UDLAP-LAFMIA, Mexico
for our collaborative construction of these slides

* This presentation was created using the content of https://www.python.org/

PYTHON CONTEXT

Python is a widely used high-level, general-purpose, dynamic programming language*:

- Its design philosophy emphasizes code readability
- Its syntax allows programmers to express concepts in fewer lines of code than possible

The language provides constructs intended to enable clear programs on both a small and large scale*

* https://en.wikipedia.org/wiki/Python_(programming_language)/

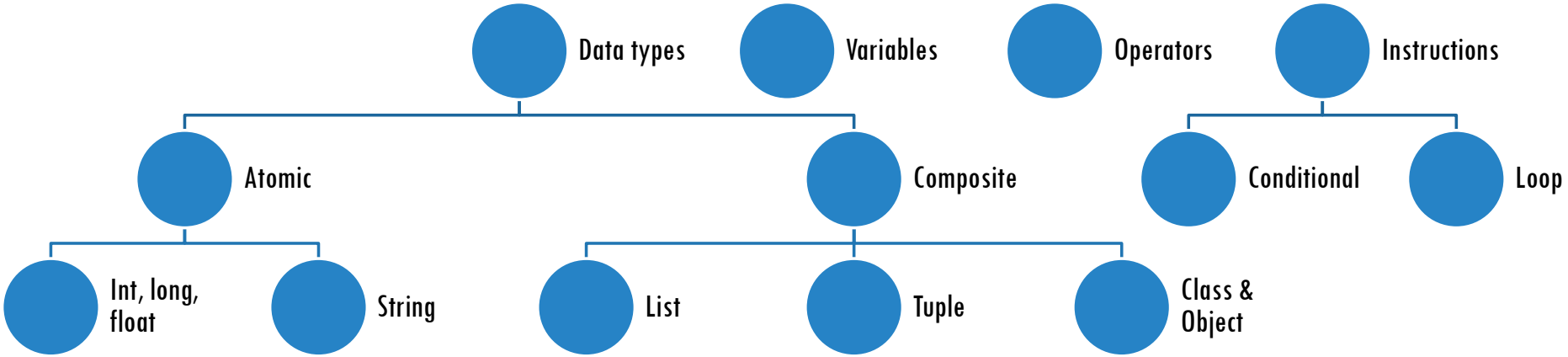
PYTHON CONTEXT

Main characteristics:

- It is an interpreted, interactive, object-oriented programming
- It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes
- It has interfaces to many system calls and libraries, as well as to various windows systems, and is extensible in C or C++
- It runs on many Unix variants, on the Mac, and on Windows 2000 and later

https://www.python.org/

RAODMAP


Data types
Variables
Operators
Instructions
Atomic
Composite
Conditional
Loop
Int, long, float
String
List
Tuple
Class & Object

DATA TYPES & VARIABLES

ATOMIC DATA TYPES

Number:

- `int`: signed integers
- `long`: long integers, they can also be represented in octal and hexadecimal
- `Float`: floating point real values
- `Complex`: complex numbers

String: contiguous set of characters represented in the quotation marks, single or double quotes

```
str = 'Hello World! '
print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])      # Prints string starting from 3rd character
print (str * 2)      # Prints string two times
print (str + "TEST")     # Prints concatenated string
```

COMPOSITE DATA TYPES: **LIST**

List: sequence of items separated by commas and enclosed within square brackets

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print (list)              # Prints complete list
print (list[0])           # Prints first element of the list
print (list[1:3])         # Prints elements starting from 2nd till 3rd
print (list[2:])          # Prints elements starting from 3rd element
print (tinylist * 2)      # Prints list two times
print (list + tinylist)   # Prints concatenated lists
```

COMPOSITE DATA TYPES: **TUPLE**

Tuple: a number of values separated by commas enclosed within parentheses

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print (tuple)                    # Prints complete list
print (tuple[0])                 # Prints first element of the list
print (tuple[1:3])               # Prints elements starting from 2nd till 3rd
print (tuple[2:])                # Prints elements starting from 3rd element
print (tinytuple * 2)            # Prints list two times
print (tuple + tinytuple)        # Prints concatenated lists

list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]

tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

COMPLEX DATA TYPES: **DICTIONARY**

Dictionary: A kind of hash table; it works like associative hashes and consist of key-value pairs

```
dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print (dict['one’])             # Prints value for 'one' key
print (dict[2])                 # Prints value for 2 key
print (tinydict)                # Prints complete dictionary
print (tinydict.keys())         # Prints all the keys
print (tinydict.values())       # Prints all the values
```

Type conversion: use the type name as a function

COMPLEX DATA TYPES: **CLASS**

```
class FourSidedShape:                          # specify the name of the class

    def __init__(self, len, wid):              # constructor of the class
          self.__length = len                  # instance private variable
          self.__width = wid

    def calculate_area(self):                  # calculate_area method
          return self.__length * self.__width

    def print_area(self):                      # print_area method
          print( "The area of the shape is " +
                          str(self.calculate_area()))
```

COMPLEX DATA TYPES: **SUBCLASS**

```
class Square(FourSidedShape):       # Inherited class enclosed in bracket

    def __init__(self, side):       # Call the parent class constructor
          FourSidedShape.__init__(self, side, side)

    def calculate_area(self):       # Call the parent class method
          return FourSidedShape.calculate_area(self)

    def print_area(self):           # over-ride the parent class method
          print( "The area of the square is " +
                          str(self.calculate_area()))
```

COMPLEX DATA TYPES: **OBJECTS**

- Create an object of FourSidedShape:

```
myShape = FourSidedShape(4, 3)
myShape.calculate_area()
myShape.print_area()
myShape.__width                    # it does not have this attribute
```

- Create an object Square:

```
mySquare = Square(4)
mySquare.calculate_area()
mySquare.print_area()
mySquare.__length                  # it does not have this attribute
```

VARIABLES

Phyton variables do **not need explicit declaration** to reserve memory space:

```
counter = 100   # An integer assignment
miles   = 1000.0    # A floating point
name    = "John"    # A string
```

Multiple assignment allows to assign a single value to several variables simultaneously or assign multiple objects to multiple variables:

```
a = b = c = 1
i, j, name = 1, 2, "John"
```

OPERATORS

OPERATORS

Arithmetic

- addition (+), subtraction (-), multiplication (*), division (/), modulus (%), exponent (**), & floor division (//)

Comparison compare the values on either sides of them and decide the relation among them

Membership test for membership in a sequence, such as strings, lists, or tuples:

- `in` evaluates to true if it finds a variable in the specified sequence and false otherwise
- `not in` evaluates to true if it does not finds a variable in the specified sequence and false otherwise

OPERATORS

`Identity` compare the memory locations of two objects:

- **`is`** evaluates to true if the variables on either side of the operator point to the same object and false otherwise

  Example: `x is y` results in `1` if `id(x)` equals `id(y)`

- **`is not`** evaluates to false if the variables on either side of the operator points to the same object and true otherwise

  Example: `x is not y` results in `1` if `id(x)` is not equal to `id(y)`

OPERATORS PRECEDENCE

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus, and minus
* / % //	Multiply, divide, modulus, and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators

INSTRUCTIONS

CONDITIONAL & LOOP

Conditional

```
if condition:
      print(True)
else:
      print(False)
```

Loop

```
for i in range(1,10,2):
      print(i)

while condition:
      print(True)
```