DS-GA 1007 | Lecture 1

Programming for Data Science

Jeremy Curuksu, PhD
NYU Center for Data Science
jeremy.cur@nyu.edu

September 11, 2023

Week 1

1. Course Information
2. Introduction to Programming in Python

Course Information

DS-GA 1007 Instructional Team

Instructor:

- Dr. Jeremy Curuksu, jeremy.cur@nyu.edu

Section Leaders and Graders:

- Jiayue (Hailey) He, jh8530@nyu.edu
- Shivam Ahuja, sa7445@nyu.edu
- Anudeep Tubati, at5373@nyu.edu

DS-GA 1007 Schedule

DS-GA 1007.001 Lecture:

- Mondays from 6:45pm-8:30pm EST
- Location: 12 Waverly Place, Room G08

DS-GA 1007.002 Lab:

- Wednesdays from 7:10pm-8:00pm EST
- Location: 19 University Place, Room 102

DS-GA 1007 Curriculum

Programming for Data Science:

- Introduction to Programming in Python
- Best Practice Programming and Software Engineering
- Program Efficiency
- Interacting with Programs
- NumPy: Array Manipulation for Scientific Computing
- Matplotlib: Data Visualization
- Pandas: Advanced Data Objects ($\times 4$)
- Git: Environment for Collaborative Programming
- Industrial Applications

DS-GA 1007 Resources

- **Lecture and lab practice code + lecture slides**
- **Python Data Science Handbook** (2017) by Jake VanderPlas
- ***The Carpentries* intro labs on Python, Linux and Git** (`software-carpentry.org/lessons/index.html`)
- Python for Data Science (2022) by Yuli Vasiliev
- The Linux Command Line (2019) by William Shotts
- Python packages used in this course have online concise high-quality doc: NumPy, Pandas, Matplotlib

Advices to Succeed in this Course

- **Attend both lectures and labs**. Lectures and labs complement each other to set you up for success
- **Practice, practice, practice**. Programming is a skillset, everyone has a unique approach, find your own!
- **Before writing a program**, define its goal and data flows
- **Break up problems into sub-problems**. Break your program up into modules that can be tested individually
- **Document your programs**. We all forget important details
- **Ask questions!**

Introduction to Programming in Python

Introduction to Programming in Python

Today topics:

- What is Programming?
- Why Programming in Data Science?
- Primitive Data Types
- Control Flow
- Compound Data Types: Tuples, Lists, Dictionaries, ...
- Manipulating Compound Data Types
- Reading/Writing Files and Examples

What is Programming?

What do Computers do?

What do Computers do?

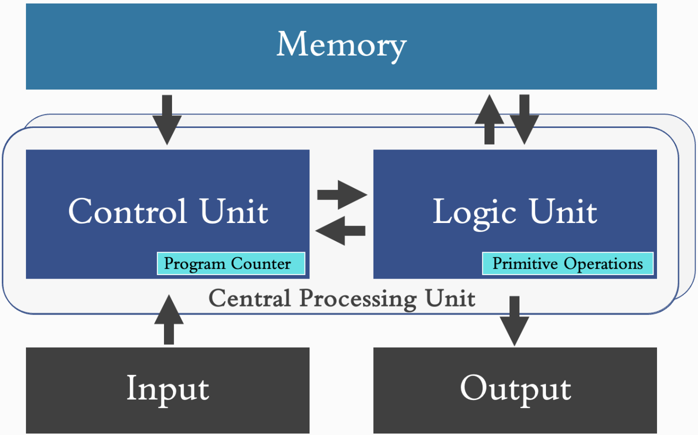
- **Perform calculations**
  - Fixed program computers
  - Stored program computers

What do Computers do?

- **Perform calculations**
  - Fixed program computers
  - Stored program computers
- **Store knowledge**
  - Declarative knowledge (statements of facts)
  - Imperative knowledge (programs)

What do Computers do?

Architecture of *stored program* computers

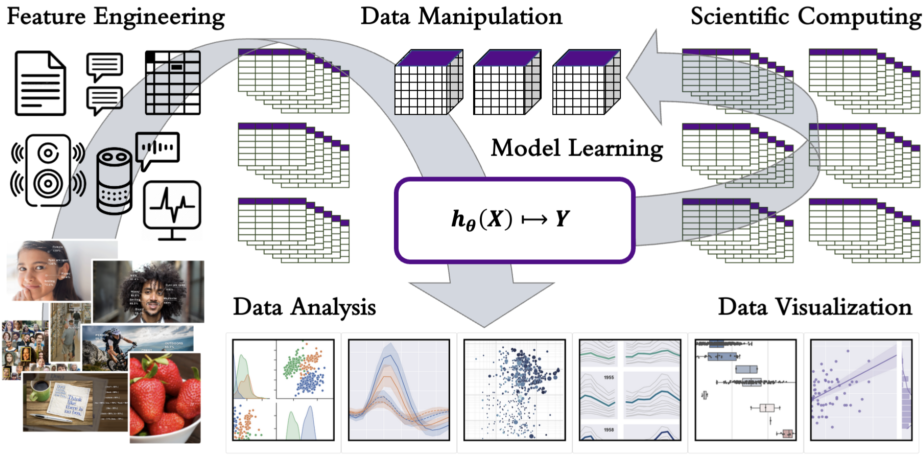


What is a Program?

- **Represent knowledge with data structures:**
  1. Primitive data types
  2. Compound data types

- **Encode an algorithm:**
  1. Instructions = Sequence of simple steps (commands)
  2. Flow of control = Specifies when each step executed
  3. Termination condition = Determine when to stop

Why Programming for Data Science?



Creating a Program

- **All instructions in a program are built from a set of primitive instructions**
  - Arithmetic and logic operations
  - Tests to change flow of control
  - Data transfers

- **A programming language offers a set of primitives**
  Anything computable in one language is computable in any other programming language

- **A special program '*interpreter*' executes instructions**

Creating a Program *in Python*

- **Data structure definition:** Evaluated by interpreter
- **Command:** Instruct the interpreter to do something
- **Data & commands can be typed interactively** into a console, or stored to file to be read later in batch
- Example:

```python
# Define the data
data = "DS-GA 1007"
# Print the data
print("Welcome to " + data)
```

Syntax and Semantics of Languages

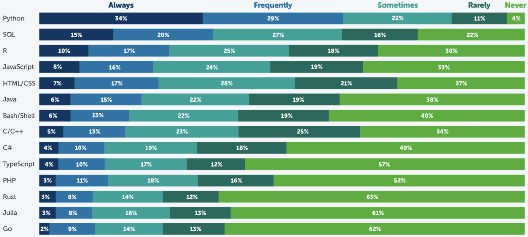
	In English (Natural Language)	In Python (Programming Language)
Primitives	Words	Numbers, Strings, Operators
Syntax	*Valid:* She likes running *Invalid:* She running likes	*Valid:* $5 + 10$ *Invalid:* $5 = 10$
Static-Semantics	*Valid:* I like pizzas *Invalid:* Pizzas like me	*Valid:* $"hi\ " + "5"$ *Invalid:* $"hi\ " + 5$
Semantics	He likes her	$x = -9 \times x + 50$

Syntax and Semantics of Languages

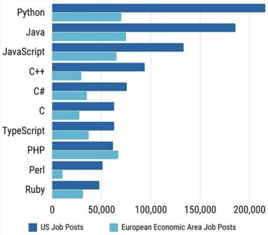
- **Syntax errors: Common and easily caught** by editor and interpreter
- **Static Semantic errors: Often but not always caught** by the interpreter, can cause unpredictable behavior
- **Semantic errors: Frequent source of problem:** program crashes, runs forever, or gives an answer but different than expected

Primitive Data Types in Python

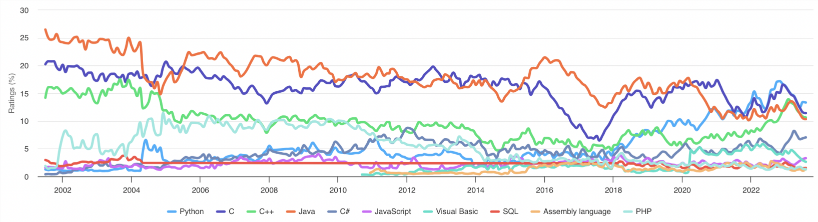
Why Python?



Poll on 4,200 data scientists from 140 countries
Source (2021): anaconda.com



Language shown on LinkedIn postings
Source (2022): codingnomads.co



TIOBE popularity index. Source (2023): tiobe.com

Python Primitive Data Types

- Programs manipulate data **objects**
- An object has a **type**: defines what the program can do to it

Primitive Python Objects		
Int	Integer Numbers	Ex: 1, 2, 3...
Float	Real Numbers	Ex: 3.14
String	Text Contents	Ex: "*Hello World*!"
Bool	Boolean Values	*True* , *False*
NoneType	Special Type	*None*

Expressions and Operators

- **Expressions**: Combinations of objects and **operators**. An expression has a value. A value has a type...
- ***"Everything in Python is an object"***

Primitive Python Operators

Arithmetic		**Comparison**		**Boolean**	
=	Assignment	==	Equality	not	Negation
+	Sum	!=	Inequality	and	Conjunction
−	Difference	>	More than	or	Disjunction (inclusive)
*	Product	>=	More or Equal		
/	Division	<	Less than		
%	Remainder	<=	Less or Equal		
**	Power				

Variables and Assignments

- An **assignment** binds a value to a **variable** name:

```
pi = 3.14
```

- The value is then **stored in memory**. It can be retrieved by invoking the variable name:

```
print(pi)
```

- A subsequent assignment **re-binds** the variable to a new value:

```
pi = 3.14159
```

- A variable can be re-bound to expressions operating on itself:

```
r = 10
area = pi * (r**2)
ncircles = 5
area = area * ncircles
```

Variables and Assignments

- **Abstraction of Expressions:** A variable name assigned to the value of an expression can be used instead of the value itself to write an algorithm *as a function of* input parameters

```
import sys
from sklearn.metrics import confusion_matrix
actuals, predictions = sys.argv[1]
m = confusion_matrix(actuals, predictions)
TP = m[1,1]
FN = m[1,0]
TP = TP / (TP + FN)# Proportion
TP = int(TP * 100) # Percentage
print("The recall is {}%".format(TP))
```

Control Flow
in Python

Control Flow: Branching, Iteration

- **Evaluate a block of code if a condition is True.**
  <condition> evaluates to a Boolean (True or False)

Branching

```
if <condition>:
    <expressions>
```

```
if <condition>:
    ...
elif <condition>:
    ...
else:
    ...
```

Iteration

```
while <condition>:
    <expressions>
```

Evaluation repeats until the condition becomes False

```
for <variable> in <>:
    ...
```

Evaluation repeats for each value taken by the variable

Control Flow: Example of Branching

- **Indentation defines blocks of code in Python**

```python
if y > 0 and x > 0:
    print("x and y are positive numbers")
if x == y:
    print("x and y are equal")
elif x < y:
    print("x is smaller than y")
else:
    print("x is larger than y")

print("What else do you want to know?")
```

Control Flow: Example of Iteration

- **Execute block of code until condition is false**

```
while x != y:
    if x < y:
        x = x + 1
    else:
        x = x - 1
print("Now x and y are equal")
```

Control Flow: Example of Iteration

- **Execute block of code until condition is false**

```
while x != y:
    if x < y:
        x = x + 1
    else:
        x = x - 1
print("Now x and y are equal")
```

...or are they?

Control Flow: Example of Iteration

- **Iterate through a preset sequence of objects**
  - With a **While** loop

```
n = 0
while n < 10:
    print(n)
    n = n + 1
```

  - Shortcut: The **For** loop

```
for n in range(10):
    print(n)
```

Creating loops with *range*

- **Create an iterable with range (start, stop, step)**
  - Loop until value is *stop* $- 1$
  - start and step are optional
  - Default values are *start* $= 0$ and *step* $= 1$

```
for n in range(10):
    <expressions>

for n in range(5, 10):
    <expressions>

for n in range(5, 10, 2):
    <expressions>
```

Breaking loops with *break*

- **Exit a loop immediately with break**
  - Skips remaining expressions in code block
  - Exits only the current ”innermost” loop

```
while x != y:
    if x < y:
        x = x + 1
    else:
        x = x - 1
    if abs(x - y) < 1:
        break
print("Now x and y are equal")
```

Compound Data Types

Compound Data Types: Tuples

- **Tuple** = Ordered sequence of information accessible by index
- Types of elements can be mixed
- A tuple is **immutable**: Values cannot be changed
- A tuple is represented with **parentheses**

```
t = ()          # Create an empty tuple
t = ("NYU",1,2,3) # Create tuple of four elements
len(t)          # Evaluates to 4 (number of elements)
t[0]            # Evaluates to "NYU"
t[1:3]          # Slice tuple, evaluates to (1,2)
t[1] = 4        # Syntax error
```

Compound Data Types: Lists

- **List** = Ordered sequence of information accessible by index
- Types of elements can be mixed
- A list is **mutable**: Values can be changed
- A list is represented with **square brackets**

```
l = []          # Create an empty list
l = ["NYU",1,2,3] # Create list of four elements
len(l)          # Evaluates to 4 (number of elements)
l[0]            # Evaluates to "NYU"
l[1:3]          # Slice list, evaluates to [1,2]
l[1] = 4        # l is now ["NYU",4,2,3]
```

Compound Data Types: Dictionaries

- **A Dictionary** stores information accessible by keys
- **Keys & values are custom data**, not ordered, type can be mixed
- **Values** can be duplicate and of any type
- **Keys** must be unique and of immutable type
- A dictionary is represented with **curly braces**

```
d = {}                  # Create an empty dictionary
rates = {'Movie 1':'A+', 'Movie 2':'B', 'Song 1':10}
rates['Movie 1']  # Evaluates to 'A+'
rates['Song 2']   # Key Error
rates['B']        # Key Error
rates['Movie 2'] = 'A+'
```

Tuple and List vs. Dictionary

Tuples and Lists

- Sequence of elements
- Look up elements by an index
- Indices have an intrinsic order
- The index is an integer

Dictionaries

- Pairs of values and keys
- Look up items (values) by other items (keys)
- Keys and values are not ordered
- The key can be any immutable type

Array and Data Frame

Arrays

Lecture 5: Array Manipulation and Scientific Computing

- Fixed-typed elements
- Look up elements by integer indexing
- Scale to large dense multidimentional data
- Fast vectorized operations

Data Frames

Lectures 7 to 10: Advanced Data Objects

- Multidimensional array with heterogeneous column types
- Missing data (NaNs)
- Labels attached to rows and columns

Manipulating Compound Data Types

Manipulating *Objects* in Python

Objects have "methods"

- **Everything in Python is an object: Lists are objects, Strings are objects, Dictionaries are objects, Arrays are objects, ...**
- Objects have data and methods (covered in Lecture 2)
- Methods are invoked by the dot notation: *object.method*()
- Examples:

```
l.append(x)       # Mutates list l by appending x
l.extend([x,y])   # Extends list l with x and y
l.pop()           # Deletes last element of list l
```

- Other functions also apply to an object depending on its type
- Examples:

```
len(l)            # Returns number of elements in list l
del(l[0])         # Deletes first element of list l
```

Operations on Strings

A string is a special type of tuple

- **Appending characters and concatenating strings**

```
request = "Give me a"
goal = "Hi" + "5"
question = request + " " + goal
```

- **Indexing characters**: Starts at 0. Last element is at index -1

```
s = "abcd"
len(s)        # Evaluates to 4 (number of characters)
s[0]          # Evaluates to "a"
s[-1]         # Evaluates to "d"
s[-4]         # Evaluates to "a"
s[1:4]        # Slice string, evaluates to "bcd
s[1] = "e"    # Syntax error
```

Operations on Strings

Slicing

- A string can be **sliced** using [start:stop:step]
- Giving only two numbers means [start:stop]
- Default step $= 1$
- Fine control possible by keeping colons and ommiting numbers

```
s = "abcde"
s[1:4]      # Evaluates to "bcd"
s[0:5:2]    # Evaluates to "ace"
s[5:1:-2]   # Evaluates to "ec"
s[:]        # Same as s[0:len(s):1]
s[::-1]     # Same as s[-1:-(len(s)+1):-1]
```

Operations on String

A string can be converted into a list

- **list(s)** returns a list where every character is an element

```
list("abcde")    # Returns ["a","b","c","d","e"]
```

- **s.split()** splits a string s on a character parameter. It splits on spaces if called without a parameter

```
l="Cook or Paint".split(" or ") # Returns ["Cook","Paint"]
l.append("Dance") # Operate on list (covered next slide)
```

- **Lists can be converted back to strings**
  s.join(l) turns a list l into a string of characters. Characters in s are added between elements of the list, but s can be empty

```
" or ".join(l)  # Returns "Cook or Paint or Dance"
```

Operations on Lists

Lists are mutable and can be nested

- **Appending elements and concatenating lists**

```
l = ["Cook","Paint"];  p = ["Run","Swim"]
lp = l + p            # lp:["Cook","Paint","Run","Swim"]
p.append("Dance")     # p:["Run","Swim","Dance"]
p.extend(l)           # p:["Run","Swim","Dance","Cook","Paint"]
```

- **Indexing and slicing**

```
l[0] = "Ubereat"      # l mutated: ["UberEat","Paint"]
lp[1:4]               # ["Paint","Run","Swim"]
lp[::-1]              # ["Swim","Run","Paint","Cook"]
l.append(p[:2])       # l:["UberEat","Paint",["Run","Swim"]]
l[2]                  # ["Run","Swim"]
p[1] = "tv"           # l:["UberEat","Paint",["Run","tv"]]
```

Operations on Lists

Aliasing

- **Aliasing lists** $(=)$ side effect: changing one changes the other!

```
warm = ["red","yellow","orange"]
hot = warm
hot.append("pink")
```

```
print(hot)
```

Output: ["*red*","*yellow*","*orange*","*pink*"]

```
print(warm)
```

Output: ["*red*","*yellow*","*orange*","*pink*"]

Operations on Lists

Cloning

- **Cloning lists** creates a new list and copies every element

```
warm = ["red","yellow","orange"]
hot = warm[:]
hot.append("pink")
```

```
print(hot)
```

Output: ["*red*", "*yellow*", "*orange*", "*pink*"]

```
print(warm)
```

Output: ["*red*", "*yellow*", "*orange*"]

Operations on Lists

Sorting lists

- **sorted** does not mutate list, must assign to variable

```
warm = ["red","yellow","orange"]
sortedwarm = sorted(warm)
```

```
print(warm)
```

Output: ["*red*", "*yellow*", "*orange*"]

```
print(sortedwarm)
```

Output: ["*orange*", "*red*", "*yellow*"]

- **sort()** mutates the list, returns nothing

```
sortedwarm = warm.sort()
```

```
print(warm)
```

Output: ["*orange*", "*red*", "*yellow*"]

```
print(sortedwarm)
```

Output: *None*

Operations on Dictionaries

Dictionaries are mutable and can be nested

- **Adding, testing an deleting entries**

```
rates = {'Movie 1':'A', 'Movie 2':'B'}
rates['Movie 3'] = 'A' # Add new entry, key must be unique
'Movie 3' in rates      # Returns True
'Movie 4' in rates      # Returns False
len(rates)              # Returns 3 (number of entries)
del(rates['Movie 3'])   # {'Movie 1':'A','Movie 2':'B'}
```

- **Extracting Keys and Values**

```
rates.keys()   # Returns iterable ('Movies 1','Movie 2')
rates.values() # Returns iterable ('A',' B')
rates.items()  # Returns (('Movie 1','A'),('Movie 2','B'))
```

Iterating over string, list, dictionary

The Pythonic way...

- **Strings**

```python
s = 'abcde'
for i in s:   # for i in range(len(s)):
    print(i)  #     print(s(i))
```

- **Lists**

```python
l = [1,2,3,4,5]
for i in l:   # for i in range(len(l)):
    print(i)  #     print(l(i))
```

- **Dictionaries**

```python
d = {1:'a',2:'b',3:'c',4:'b',5:'a'}
for k in d.keys():
    print(d[k])
```

Example with string, list, dictionary

Find frequency of each word in a song:

```
lyrics = "I heard there was ... Hallelujah".split()
d = {}
for word in lyrics:
   if word in d:
      d[word] += 1
   else:
      d[word] = 1

print(d['Hallelujah'])
```

Example with string, list, dictionary

Find frequency of each word in a song:

```
lyrics = "I heard there was ... Hallelujah".split()
d = {}
for word in lyrics:
   if word in d:
      d[word] += 1
   else:
      d[word] = 1

print(d['Hallelujah'])
```

Output: 25

Read Input

- **Prompt user for input.** Binds entry to variable

```
song = input("Write a song")
word = input("Type a word")
```

Read Input and Print Output

- **Open file, read file, print to file**

```
infile = open("input.dat","r")
outfile = open("output.dat","w")

lines = infile.readlines()
print(lines[-1]) # Print last line of input file

print("Occurences of Hallelujah:", file=outfile)

for line in infile:
   if("Hallelujah" in  line): outfile.write(line)
```

Read Input and Print Output

- **Read dictionary input files**

```
import json
dictcontents =  json.load(open('dictfile.json'))
```

- **Format string output**

```
s = input("Type a sentence: ")
l = s.split()
n = len(l)
print('Count{0:>8}\n First{1:>8}'.format(n,l[0]))
```

Execute and Interface with Program

- **Demo this code:**

```
song = open("lyrics.txt","r")
word = input("Type a word: ")
d = {word: 0}
for line in song:
      if word in line:
            d[word] += 1
print('The word {} appears {} times in this song'
      .format(word,d[word]))
```

Execute and Interface with Program

- **Demo this code:**

```
song = open("lyrics.txt","r")
word = input("Type a word: ")
d = {word: 0}
for line in song:
      if word in line:
            d[word] += line.count(word)
print('The word {} appears {} times in this song'
      .format(word,d[word]))
```

Thank you!