Introduction to (Python) programming

Terence Parr
MSDS program
University of San Francisco

What is programming / coding?

- Literally: Creating a set of instructions for a computer to execute
- First we construct a sequence of abstract operations, sometimes called an *algorithm* or *workplan*, that performs the desired task
- Then we translate these abstract operations to concrete and precise instructions

  Say hello ☞ `print("hello")`

- These instructions must follow the grammatical structure of a programming language, such as Python
- Each instruction typically solves a piece of the problem
- The emergent behavior of the program solves our task

What's the hard part?

- Programming is mostly about converting "word problems" (project descriptions) to algorithms or work plans
- We immediately think about programming languages because we express ourselves using specific language syntax but…
- Programming is more about *what* to say, and in what order, rather than *how* to say it
  - You'll eventually get fast at Python coding and using libraries
  - It'll always be harder to design a sequence of steps that solves a data science problem (or other) than it is to code
  - I remember being confronted with my first programming task (using BASIC in 1979!) and drawing a complete blank even though I knew BASIC syntax
- Don't worry: we will study lots of patterns and strategies as aids

Concrete first steps

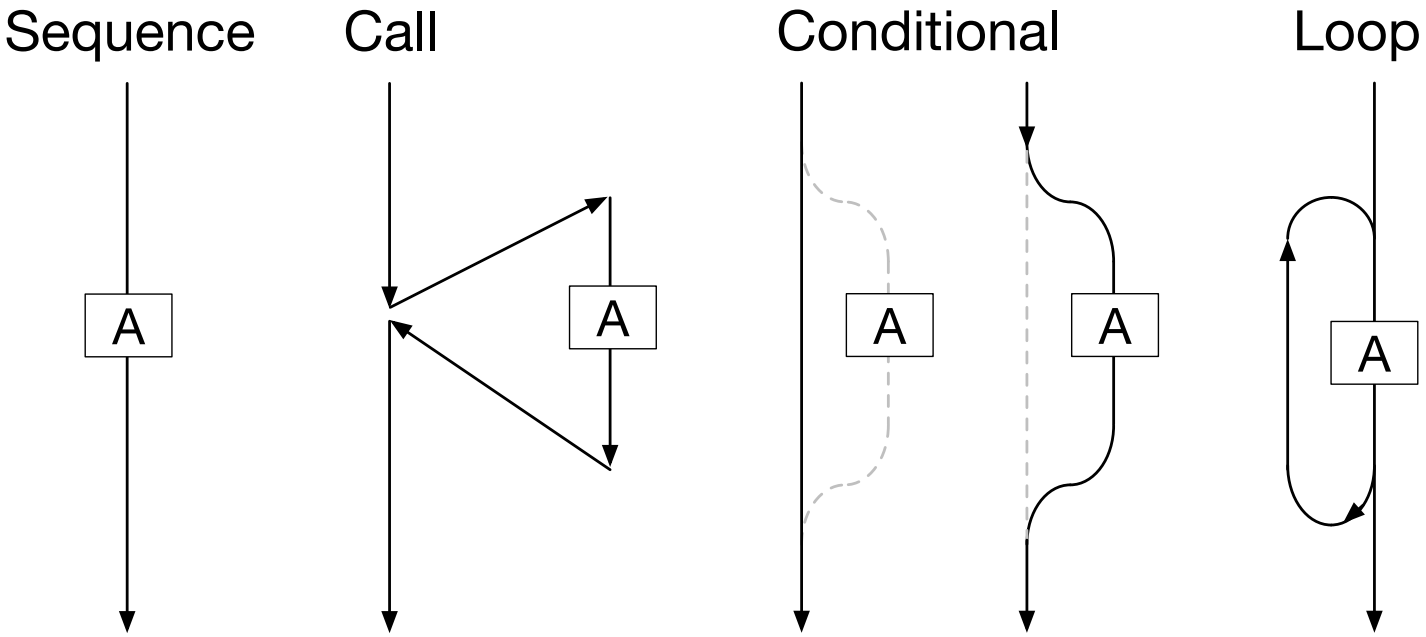
- Programming is more about design, rather than coding details, but it's much easier to learn programming by actually speaking Python (e.g., we begin learning a foreign language by memorizing a few key phrases like "*May I have a beer?*")
- Let's get started by looking at the key pieces we have at our disposal and then we can learn some basic Python and write some simple programs

Most important programming concepts

- Order of operations (*control-flow*)
- Representing data in memory
- Batch execution vs interactive execution
- Aggregating instructions into reusable methods
- Aggregating instructions and methods into modules (.py files)
- Object-oriented (OO) programming (aggregating data, methods)

Key concept: order of operations

- Order is critical
  Example: get license, buy car, drive car



Key language constructs

```
42 3.14
"string" 'string'
[ expr , expr ,… ]

var = expr

func ( expr , expr ,… )

expr . func ( expr , expr ,… )

[ expr for var(s) in elements ]

[ expr for var(s) in elements if condition ]
```

```
if condition :
    statement(s)
else:
    statement(s)

while condition :
    statement(s)

for var(s) in elements :
    statement(s)

import package

import package as alias
```

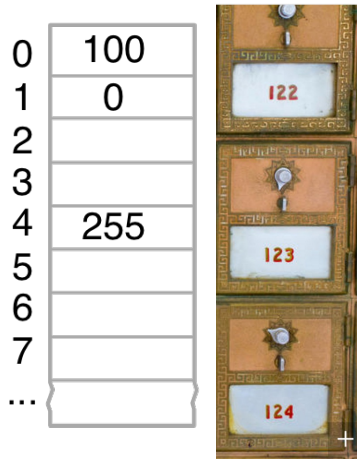
UNIVERSITY OF SAN FRANCISCO

Interactive demos via pythontutor.com

- Let's observe the control-flow using our key syntax constructs:
  - generate some output
  - assignment creates and alters variables
  - types matter, operators are overloaded
  - simple conditional execution
  - else-clause
  - simple loop that updates variable
  - demo loop for powers of two

Programmer's view of memory

Representing data in memory



- RAM is a sequence of discrete slots where we can stick values 0..255 called bytes; made up of 8 bits as 2^8=256
- Numbers, music, videos, text are all decomposed into one or more of these discrete bytes
- Data elements have *values* and *types*
  - integer 32
  - string "hello"
  - floating point real number 3.14159
- A special element called a *pointer* or *reference* refers to another element; like a phone number "points at" a person but isn't a person
- We build data structures by combining and organizing data elements with references

Key size metrics

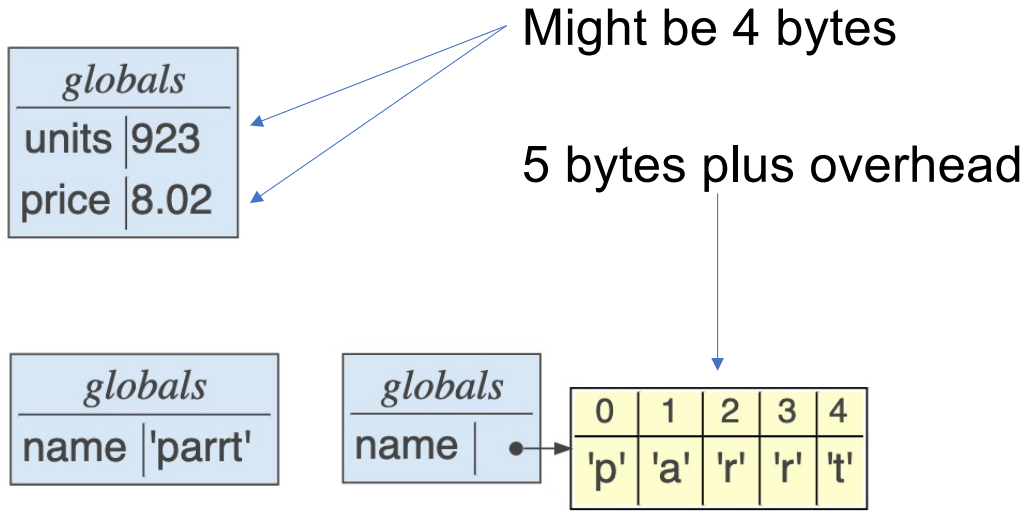
- Know these units; as data scientists, you need to know whether a data set fits in memory or whether it fits on the disk or even how long it will take to transfer across the network
  - Kilo. 10^3 = 1,000 or often 2^{10} = 1024
  - Mega. 10^6 = 1,000,000
  - Giga. 10^9 = 1,000,000,000
  - Tera. 10^{12} = 1,000,000,000,000
- On an 80 megabits/second network you can transfer 10 megabytes/second; 100M file transmits then in 10 seconds

Programming language view of memory

- Dealing with bytes is tedious; we prefer to group bytes into higher-level values, such as numbers and strings

```
units = 923
price = 8.02
```

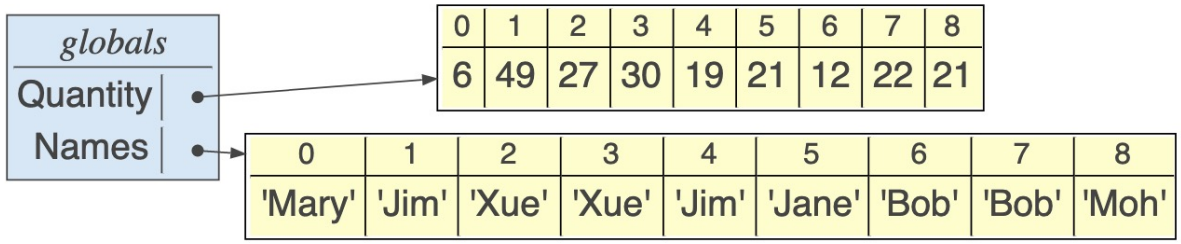
```
name = "parrt"
```



Lists of data elements

- Most common *data structure* is the **list**, which is just a sequence of data elements or other data structures

```
Quantity = [6, 49, 27, 30, 19, 21, 12, 22, 21]
Names = ['Mary', 'Jim', 'Xue', 'Xue', 'Jim', 'Jane', 'Bob', 'Bob', 'Moh']
```



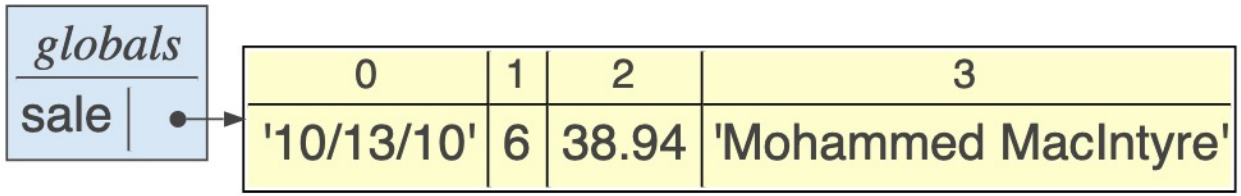
Quantity
6
49
27
30
19
21
12
22
21

- Indexed from 0 not 1 and list vars point at a chunk of memory holding the list elements contiguously (preserving the sequence order)
- Access elements with index operator; e.g., **Names[0]** is '**Mary**' and **Quantity[4]** is 19

Hetergeneous lists

- Elements can have different types:

```
sale = ['10/13/10', 6, 38.94, 'Mohammed MacIntyre']
```

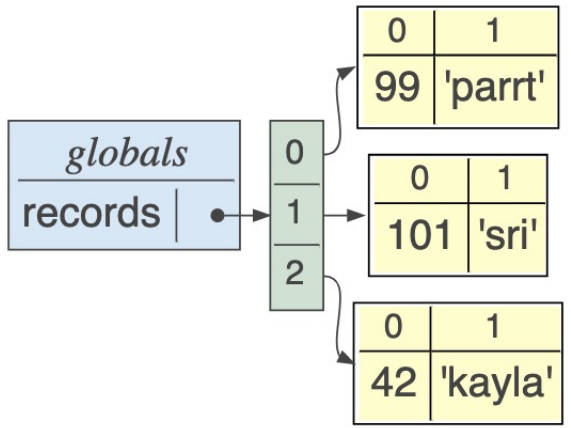


- Heterogeneous lists used to group bits of information about a particular entity or observation

List of lists

- In this case, records points at a list of three items that happen to be lists also
- [experiment via pythontutor]

```
records = [[99, 'parrt'],
           [101, 'sri'],
           [42, 'kayla']]
```

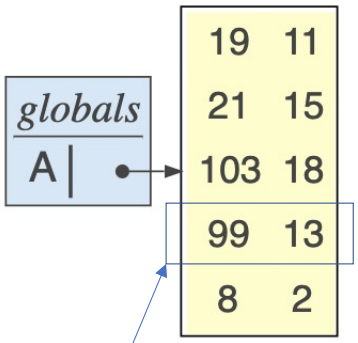


Experiment via pythontutor

Matrices as lists of lists

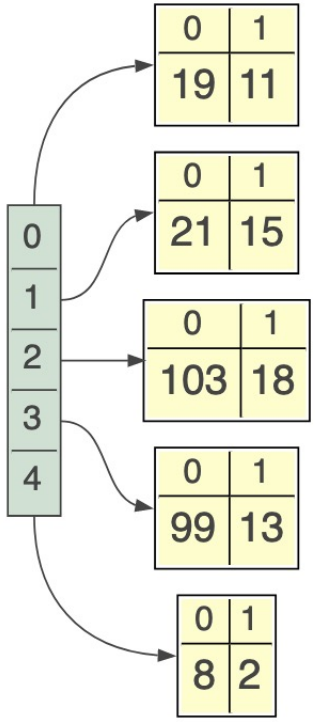
- A matrix is a list of rows; a row is a list of numbers
- We think of it like this:

```
A = [[19,11],
     [21,15],
     [103,18],
     [99,13],
     [8,2]]
```



Row 4 is A[3]

But, it's actually represented like:



Sets

- A set is just an unordered, unique collection of elements; here is an example using integers:
  `ids = {100, 103, 121, 102, 113, 113, 113, 113}`
- We can do lots of fun set arithmetic:

```
{100,102}.union({109})
```

```
{100, 102, 109}
```

```
{100,102}.intersection({100,119})
```

```
{100}
```

Tuples

- A tuple is an immutable list and uses parentheses and not square brackets for notation
- Tuples are often used to group related elements:

```
me = ('parrt',607)
userid,office = me
print(userid)
print(office)
print(me[0], me[1])
```

```
parrt
607
parrt 607
```

Experiment in pythontutor

Dictionaries

- If we arrange two lists side-by-side and kind of glue them together, we get a **dictionary**
- Dictionaries map one value to another, just like a dictionary in the real world maps a word to a definition
- Here are two sample dictionaries:

```
movies = {'Amadeus':1984, 'Witness':1985}
```

'Amadeus' → 1984
'Witness' → 1985

- Index by key to get value; e.g., `movies['Amadeus']`

title	year
A Soldier's Story	1984
Places in the Heart	1984
The Killing Fields	1984
A Passage to India	1984
Amadeus	1984
Prizzi's Honor	1985
Kiss of the Spider Woman	1985
Witness	1985

Experiment in pythontutor

Dictionary keys and values

- We can split a dictionary apart to get the keys and values:

```
print(movies.keys())
print(movies.values())
```

```
dict_keys(['Amadeus', 'Witness'])
dict_values([1984, 1985])
```

- Note: this uses the notation *object.function()*, which you can think of as *function(object)*; we'll learn more about this later

Iterating through a dictionary

- We can walk the keys/values of a dictionary with a *for-each* loop

```
movies = {'Amadeus':1984, 'Witness':1985}
for m in movies: # walk keys
    print(m)
```

```
Amadeus
Witness
```

```
for m in movies.values(): # walk values
    print(m)
```

```
1984
1985
```

More on looping next!

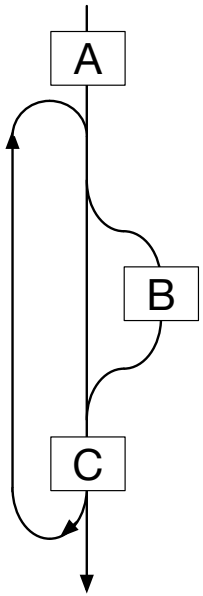
```
for (key,value) in movies.items():
    print(f"{key} -> {value}")
```

```
Amadeus -> 1984
Witness -> 1985
```

More on looping

Combined conditional / loop

- Now that we have some basic Python skills, let's look at more complicated loops starting with a combination:




```
i = 1
while i <= 6:
    if i==3:
        print("Halfway!")
    i = i + 1
```

Step through code at pythontutor.com

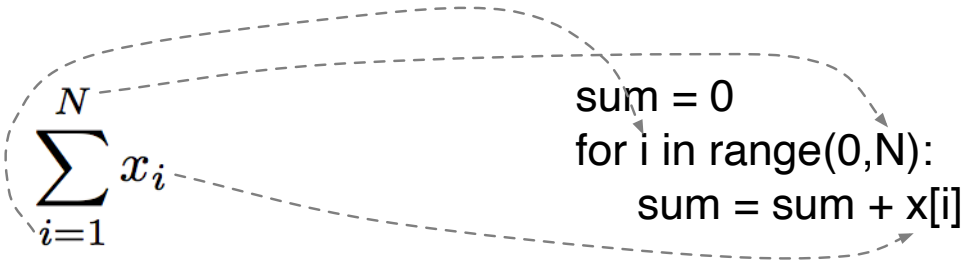
For-loops

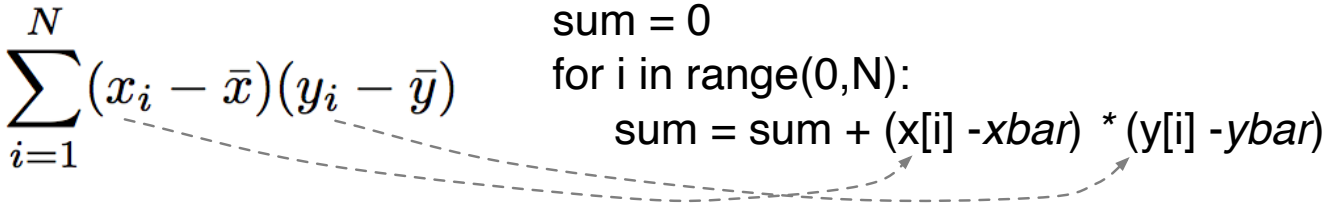
```
for var(s) in elements :
    statement(s)
```

- range loops
- for-each loops
- loop with enumerate()
- watch row var iterate through list-of-list rows
- indexed loop using range
- zip'd loop

Translating formulas

- Sigmas become accumulator range-loops (recall indexed from 0)





List comprehensions

- Making new lists from (optionally filtered) sequences, elements

```
[ expr for var(s) in elements ]

[ expr for var(s) in elements if condition ]
```

- comprehensions on lists of strings
- comprehensions on lists of numbers