Math 260: Python programming in math

Fall 2020

Fundamentals (Part 1):
Course intro, language essentials

Section 0: Course intro

About the class

- Instructor: Jeffrey Wong
- Office: on Zoom (OH/ask for appt)

My interests:

- Applied math - modeling
- Thin liquid films and instabilities
- Numerical simulation (PDEs)

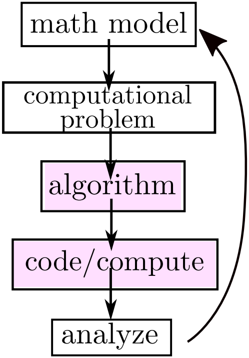
About the class

What is scientific computing?

- mix of modeling/math/computing
- Identify a computational problem
- Use this as an ‘experiment’ to get insight into problem
- Or, computation may be the goal

Our focus:

- Translating algorithm to code
- Using python to compute



Discussion question: what are some applications of interest?
Interesting examples of programming in math you've encountered

Break: introductions

What is (good) programming?

Programming languages are languages!

- syntax (vocabulary/grammar) is a first step
- Fluency: Learn to write elegant, effective code
- Ask: Could colleagues and future you understand/use your code?

Other points:

- Code often has to adapt - write so this is not painful
- Expect bugs - write to protect yourself from errors

What matters for scientific computing?

- Often smaller, specialized code
- Efficiency and accuracy really matter
- Usability: code shared/used by a team of scientists
- Intertwined with numerical analysis, mathematics

How to get started

Set up python:

- Follow getting started guide (install python etc.)
- `jupyter` (allows text/code together)

Practical advice:

Bugs happen; code is hard to fix!

- Expect things to go wrong; plan for time to fix them.
- Use other students/me as resources (ask for help!)

Review course guidelines: Some highlights...

- Collaboration is good! You may work with others on code.
- Write the 'final product' yourself. (list who you worked closely with)
- Okay to use outside sources (typical thing to do for coding!)...

  ...but avoid looking up 'the answer' online
- If using (direct) code from outside sources, cite briefly in a comment

What is Python?

Advantages:

- Easy to use, elegant by design
- Good support for computation (`numpy`, `scipy`, `pandas`)
- 'interactive' output
- popular, free (good for collaboration)

Disadvantages:

- Not as fast as C, FORTRAN (compiled)
- Less 'low-level' control: memory management...
- Less structure for building 'big' software (vs. C#, C++...)

Section 1: Language basics

A first program

Fibonacci numbers:

$$F_k = F_{k-1} + F_{k-2}, \qquad F_0 = F_1 = 1$$

Goal: write a **function** `fib_list(n)`:

- Input: integer $n \geq 2$
- output: list of values $(F_0, \cdots, F_n)$

```
def fib_list(n):
    seq = ...
    # ... code ...

    return seq
```

- Running code defines the function
- Call using `f = fib_list(n)`

A first program

File `fib.py`:

```
def fib_list(n):
    '''Computes F_0, F_1, ..., F_n'''
    fibs = [0]*n
    fibs[0] = 1
    fibs[1] = 1

    for k in range(2, n):
        fibs[k] = fibs[k-1] + fibs[k-2]

    return seq

if __name__ == '__main__':
    n = 10
    f = fib_list(n)
    print(f)
    print('F_{} = {}'.format(n, f[n-1]))
```

`fib_list`:

- Initializes `fibs` $= [0, 0, \cdots, 0]$
- For loop: sets k-th element of `fibs`
- **returns** the list

`main`:

- python executes 'main' when run
- Simple print of list f $([1, 1, \cdots, 55])$
- Formatted print: string plus variables

Functions and main

- Python console tracks current session
- Use code by (i) manual console entry or (ii) writing functions to call

```
>>> n = 10
>>> f = fib_list(n)
...etc...
```

$\Longleftrightarrow$

```
def foo():
    n = 10
    f = fib_list(n)
    ...etc...

>>> foo()
```

- main: defines what python does when the program is run

```
if __name_ _== '__main__':
    # code for main...
```

- 'Running' code (`python myfile.py`) **defines** functions, calls **main**

Best practices:

- Algorithm code goes in non-main functions; main 'calls' the code
- Best practice: avoid 'global' variables (outside functions)
- (Almost) all code should live in a function

For loops

- For loop: iterate over a list

```
for item in list:
    do_stuff(item)
    ...
```

- For loops can loop over any list:

```
names = ['apple','tomato','carrot']
for name in names:
    if is_vegetable(name):
        print('{} is a vegetable!'.format(name))
```

Looping over an integer range:

- `range(m,n)` represents $(m, m+1, \cdots, n-1)$ (****not inclusive on the right****)
- `range(m,n,k)` keeps every k-th number
- $k < 0 \rightarrow$ reverse order

```
def countdown(n):
    for k in range(n, 0, -1):
        print(k)
    print('Go!')
```

For loops

- `range` $\neq$ `list` (smarter representation!)
- Key point: **range is more efficient**

```
r = list(range(2,10000,2))
sum = 0
for j in r:
    sum += j
```

vs.

```
sum = 0
for j in range(2,10000,2):
    sum += j
```

Best practices

- use range when possible
- Common ‘integer’ loop vars: `i, j, k, m, n`

Conditionals

- **boolean** variable: true or false (python: `True`, `False`)
  (as integers: `True` = 1 and `False` = 0)
- Basic if structure:

```
if condition:
    ...
elif condition
    ...
(more elifs)
    ...
else
    ...
```

```
if n > 10:
    print('too large'!)
elif 5 < n and n < 9
    print('not too large!')
else
    print('too small!')
```

Boolean operations:

- Equality: `a==b` returns `True` iff a, b have equal **values**
- Others: `<`, `>` and `<=`, `>=` and `!=` (not equal)
- operators: `x and y`, `x or y`, `not x`
- Be careful with order of operations:

`not n>=1 or n<0` vs. `not(n>=1 or n<0)`

While loops

- While loop: iterate while condition is true

```python
while condition:
    ...
```

- while vs. `for` $\implies$ use `for` when the range is known
- `break` (ends loop) and `continue` (skips to next iteration)

```python
i = 1
sum = 0
while sum < 10 and i <= n:
    sum += i**2
    i += 1
```

```python
for obj in dataset:
    if already_processed(obj):
        continue
    process(obj)
    ... more code ...
```

Best practices

good while conditions are preferred over break:

```python
# no break
i = 1
sum = 0
while sum < 10 and i <= n:
    sum += i**2
    i += 1
```

```python
# with break
while i <= n:
    sum += func(i)
    if sum >= 10:
        break
    i += 1
```

What is a list?

```
arr = [0]*n
```

arr	0	1	2	...	n-3	n-2	n-1
				...	-3	-2	-1

- List: array-like object in python (indexed ‘list’ of items)
- `arr[j]` is the j-th item; `arr[-1]` returns last
- `arr[j] = x` sets j-th item to `x`
- Can hold any type of object: `[1,'snake',[1,2]]`
- `[1,2]*2` $\to$ `[1,2,1,2]` and `[1,2] + [3,4]` $\to$ `[1,2,3,4]`
- and much more...

Initialization:

- Allocates space in memory, sets initial values
- Explicit: `arr = [1,177,3]`
- Length n: `arr = [x]*n` (length n, all entries x)
- **list comprehensions** (more later): `arr = [2*x for x in range(n)]`

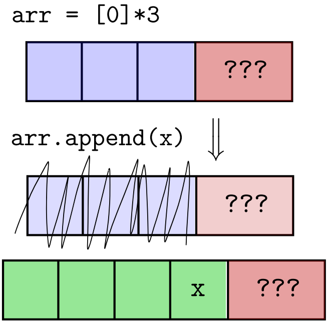
Resizing lists

Lists **do not** have a fixed size:

- Size of a list: `len(arr)`
- Add items with `append` (one element)
- use `extend` for multiple elements

```
x = [1,2,3]
x.append(4)
x.extend([5,6]) # x is [1,2,3,4,5,6]
```

- Beware: resizing a list is not free!



  - Resize must 're-allocate' (reserve new space) and move the data!
  - (Python lists are a bit smarter than this)

Best practices

Pre-allocate: initialize with correct size, once (avoid resizing!)

```
x = [0]*n  # fast!
for k in range(n):
    x[k] = k
```

vs.

```
x = []
for k in range(n):
    x.append(k)  # slow!
```

Variables: primitive types

Each variable has a type (check with `type(x)`)

'Primitive' (fundamental) types:

`int`: integer (*)

`float`: floating point number (more on this later) (*)

`boolean`: `True` (1) or `False` (0)

`string`: sequence of characters (`'blah'` or `"blah"`)

- Be careful with explicit ints vs. floats:

```
>>> type(1) # returns: int
>>> type(1.0) # returns: float
>>> type(2*3) # returns: int
>>> type(2*1.0) # returns: float
```

- `int` times `float` returns a float; `a//b` is always a float
- 'int' division: `a//b` returns a/b rounded down to an int
- (*) int has no max size; float has a min/max size

Variables and references

What is a **variable**?

- Analogy: a ‘cookie’ is that thing

  A box of cookies may have a label indicating it contains cookies

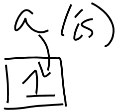
- Two perspectives on variables:

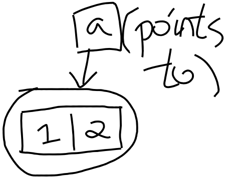
  The variable ‘is’ that thing

  `a = 1` means that `a` ‘is’ one

- **or** the variable ‘refers to’ or ‘points to’ the thing:

  `a = [1,2]` means `a` refers to the data `[1,2]`





What does this mean for python?

- A variable in python is a **name** that refers to **data**
- the **data** is the actual information in memory
- A **reference** is a thing that points to data

  (but is not the data itself, e.g. `a` in `a = [1,2]`)

Fundamental point: when does python use each perspective?

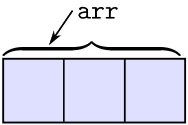
Variables: mutability

Definition (mutability)

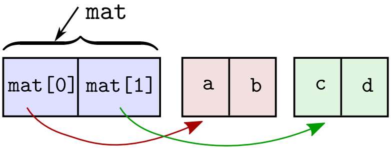
An object is **mutable** if its contents can be modified using that object. Otherwise, it is is called **immutable**.

- Primitive types are **immutable**
- Lists are **mutable** (contents can change)

- ‘mutable objects act as references to data
  (they point to some location in memory)
- Data with no reference is ‘freed’ (memory can be re-used)



`arr = [0]*3`



`mat = [[a,b],[c,d]]`

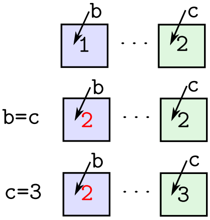
Mutable: `arr[1] = 2` changes the contents of `arr`
Immutable: if a=2 then it cannot be changed to three...
...unless it is replaced entirely (a=3)

What does a=b ('assignment') do?

- **assigns** the RHS to the LHS
- different behavior for immutable/mutable objects!

immutable assignment:

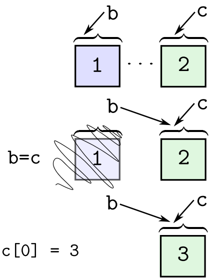
```
b = 1
c = 2
b = c
c = 3
```



Result: b is 2 and c is 3

mutable assignment:

```
b = [1]
c = [2]
b = c
c[1] = 3
```



Result: b and c both [3]

Variables: mutability

Rules for a=b:

- mutable objects are assigned 'by reference'
  - The LHS is set to point to the same data is b
  - a and b become two names for the **same data in memory**
- immutable objects are assigned 'by value'

  A copy of the RHS data is created; LHS now is set to it

Rules for **functions** are similar (how are inputs passed?):

```
def func(a):
    return 2*a

b = 3
c = func(b)  # c is 6, b is 3
```

- mutable: 'passed by reference'

  - func gets a *local* reference to the input data

  -func can modify the data

```
def func(arr):
    arr[0] = 3

a = [1, 2]
func(a)  # a is now [3, 2]
```

- immutable: 'passed by value'

  -func gets a new copy of the data

  - changes do not modify the input

Functions and mutability

- **Key point:** *mutable* types are passed ‘by reference’
- *Immutable* types (int, float, ...) are (effectively) copied (local values)

```
def doubler(x):
    x *= 2
    # value of `local' x lost

y = 5
doubler(y)
# y is still 4
```

x has y’s value (not data)

```
def doubler(arr):
    for k in range(len(arr)):
        arr[k] *= 2

v = [1,2]
doubler(v)
# v now [2,4]
```

doubler has reference to v’s data

Caution: shadowing

- ‘Shadowing’: using the same name for both outer/inner variable
- Common shorthand... ...sometimes a bad idea
- More on this later (scope rules)

```
def doubler(x):
    x  *= 2

# in another part...
x = 4
y = doubler(x)
```

Variables: mutability

More examples:

```
#Example 1:
b = [1]
c = [2]
b = c
c = [3]
# What are b and c?
```

```
#Bonus example:
b = [1,2]
b[0] = b
b[0][1] = b[0]
#what is b?
```

```
#Example 2:
row = [1,2]
b = [row,row]
b[0][0] = 7
#what is b?
```

```
#Example 3:
row = [1,2]
b = [[0,0],[0,0]]
for k in range(2):
    b[0][k] = row[k]
    b[1][k] = row[k]
b[0][0] = 7
#what is b?
```

Scope

Definition

A variable's **scope** is the region of the code where it can be accessed. Such a region is called a **namespace**.

- A program loads into a **global namespace** (the largest) the python console has access to this namespace
- Functions have their own **local** namespaces
- local variables cannot be seen outside their namespace

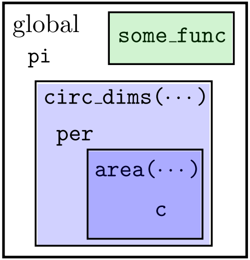
```
pi = 3.141592654 # global

def circ_dims(radius):
    per = 2*pi*radius # local

    def area(radius):
        c = 1 # (unused)
        return per*radius/2

    return area(radius), per

def some_func():
    print(per)  # fails!
```



pi seen by all

per seen by circ_dims, area only

c seen by area only

Scope

- When a variable leaves its scope, it is deleted (memory is **freed**)
- $\Longrightarrow$ local variables are not accessible outside their function
- **shadowing:** Two variables in outer/inner scopes have the same name

```
def greeter(name):
    # local var. `name'
    print('Hello ' + name)
    # local `name' destroyed

name = "Albert"
greeter(name)
```

```
def doubler(x):
    # local reference x
    for k in range(len(x)):
        x[k]*=2

x = [1,2]
doubler(x)
```

Scope

Quick note: loops do **not** have their own scope (only functions)

```python
sum = 0
for k in range(5):
    sum += k

print(sum) # 10
print(k) # 4
```

```python
sum = 0
k = 0
while k < 5:
    sum += k
    k += 1

print(sum) # 10
print(k) # 5
```

(k exists after the loop is done!)

Lists vs. tuples

Tuples

- fixed size version of a list, **immutable type**

```
arr = [1,2,'a']
tup = (1,2,'a')
arr[0] = 1 # ok
tup[0] = 1 # error! (no assignment)
```

- Good for holding small groups of fixed data; used by **return**:

```
def func(x):
    a = 2*x
    b = x + 2
    return a, b, # ---> tuple (a,b)

c, d = func(2) # set returns individually
t = func(2) # return as tuple
```

- Tuples: can be **optimized** better by python (fixed size)
- Lists: **much more** flexible

Input and output (I/O), the basics

Important: **formatted** output is more clear (not just values)

```
# assume N = 5 and fib[N] = 8
print(N, fib[N]) # prints 5 8
print("Fib {0} = {1}".format(N, fib[N])) # Fib 5 = 8
print("Fib {} = {}".format(N, fib[N])) # Fib 5 = 8
pval = 3.141592654
print("pi = {:.4f}".format(pval)) # pi = 3.1416
print("{:.2e}".format(pval*1e-8)) # 3.14e-8
```

- curly brace syntax: {`label:format`}
- blank label uses order listed in `.format(...)`
- if `label` is k, it uses the k-th variable in `.format(...)`
- Lots of formatting codes! (Look them up)

  Notable: .xf (float, x digits) and .xe (x digits, sci. notation)

Input and output (I/O), the basics

Python 3.6+ shorthand: f-strings...

- `f"string"` defines a 'formatted string' (f-string)
- For an f-string, 'labels' in braces can be the variable:

```
temp = 15.2
unit = 'Celsius'
print(f"It's {temp:.0f} deg. {unit}.") #It's 15 deg. Celsius.
print("It's {:.0f} deg. {}".format(temp, unit)) #(same)
```

(i.e. the variable can be 'plugged into' the braces)

```
pval = 3.141592654
print(f"pi equals {pval:.4f}") #prints 'pi equals 3.1416'
```

- Equivalent to `format` (just shorter)

Input and output (I/O), the basics

User input:

```
name = input('What is your name?')
print('Hi, {}!'.format(name))
```

- Use `var = input(message)` to query user input
- Three options for user input:
  i) Console input: query with `input`
  ii) File input: read from a settings/input file (covered soon!)
  iii) program arguments (passed when run), e.g.

     ```
     python prog.py 1 2 moo
     ```

  iv) no run-time input: set values by editing code in main
- (iv) is convenient for short examples
- Best practice: write code that can be used without being changed