Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

April 19, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. Functions in `Python`
2. Complexity of algorithms
3. Time complexity in `Python`.
4. `NumPy` Basics
5. `NumPy` numerics
6. Homework 2

Function: basic functions

Functions are defined using `def`, and arguments can be defined with or without default values.

```
def  printXYZ(x, y=2, z=1):
    print("x={x}, y={y} and z={z}")

```

```
printXYZ(1)
## x=1, y=2, z=3
printXYZ(1, z=-1)
## x=1, y=2, z=-1
printXYZ("abc", y=True)
## Will this work?

```

```
printXYZ(z=-1, x=0)
##  x=0, y=2, z=-1
printXYZ()
## ??
```

Functions: `return` statements

Functions must explicitly include a `return` statement to return a value.

```
def f(x):
  x**2
f(2)
##
type(f(2))
## <class 'NoneType'>
```

```
def g(x):
  return x**2
g(2)
## 4
type(g(2))
## <class 'int'>
```

```
def is_odd(x):
  if x % 2 == 0: return False
  else:          return True
is_odd(2)
## False
is_odd(3)
## True
```

Functions: multiple return values

Functions can return several values using a tuple or list.

```python
def f():
  return (1,2,3)

f()
## (1, 2, 3)
```

```python
def g():
  return [1,2,3]

g()
## [1, 2, 3]
```

If multiple values are present and not in a sequence, then it will default to a tuple,

```python
def h():
  return 1,2,3

h()
## (1, 2, 3)
```

```python
def i():
  return 1, [2, 3]

i()
## ?
```

Function: doc strings

A doc string is a short, concise summary of the object's purpose. Doc strings are specified by supplying a `string` as the very line in the function definition.

```
def f():
  "Hello."
  pass
f.__doc__

## 'Hello.'

def g():
  """This function does
  absolutely nothing.
  """
  pass
g.__doc__

## 'This function does\n    absolutely nothing.\n    '

```

Function: Variadic arguments

If the number of arguments is unknown, it is possible to define variadic functions

```
def paste(*args, sep=" "):
  return sep.join(args)

paste("A")
## 'A'

paste("A","B","C")
## 'A B C'

paste("1","2","3",sep=",")
## '1,2,3'
```

Function: variadic arguments

Positional and/or keyword arguments

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
-----------    ----------     ----------
|              |                  |
|        Positional or keyword   |
|                                 - Keyword only
-- Positional only

```

Example:

```
def f(x, /, y, *, z):
  print(f"x={x}, y={y}, z={z}")

f(1,1,1)
## Error in py_call_impl(callable, dots$args, dots$keywords)
: TypeError: f() takes 2 positional arguments but 3 were
given
##
## Detailed traceback:
##   File "<string>", line 1, in <module>

```

Functions: anonymous and annotations

Anonymous functions

```
def f(x,y):
  return x**2 + y**2
f(2,3)
## 13
type(f)
## <class 'function'>
```

```
g = lambda x, y: x**2 + y**2
g(2,3)
## 13

type(g)
## <class 'function'>
```

Function annotations (type hinting)

```
def f(x: str, y: str, z: str) -> str:
  return x + y + z

f.__annotations__
## {'x': <class 'str'>, 'y': <class 'str'>, 'z': <class 'str'>, 'return': <class 'str'>}
```

Functions: small exercises

1. Write a function, `kg_to_lb`, that converts a list of weights in kilograms to a list of weights in pounds (there a 1 kg = 2.20462 lbs). Include a doc string and function annotations.
2. Write a second function, `total_lb`, that calculates the total weight in pounds of an order, the input arguments should be a list of item weights in kilograms and a list of the number of each item ordered.

Complexity of algorithms

- Tool describes an algorithm's complexity, usually in time but also in space/memory.
- The largest term involving n in that function.
- Different notations: worst-case or upper bound: Big-O ($O(n)$), best-case or lower bound: Big-Omega ($\Omega(n)$), average-case: Big-Theta ($\Theta(n)$)
- complexity ignores constants and scaling factors

Complexity	**Big-O**
Contant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Quasilinear	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

...but processors are getting faster and memories cheaper...

1. Complexity is different to the actual execution time.
2. The execution time depends on processor speed, instruction set, disk speed, compiler, etc.
3. Complexity is about the algorithm, the way it processes the data to solve a given problem. It's a software design concern at the "idea level."
4. An inefficient algorithm may seem efficient on high-end hardware. With a large input, the limitations of the hardware will become apparent.

Complexity of algorithms

Are there techniques to figure out the complexity of algorithms?

- Instead of looking for exact execution times, we should evaluate the number of high-level instructions with respect to the input size.
- A single loop that iterates through the input is linear.
- A loop within a loop, with each loop iterating through the input, is quadratic.
- A recursive function that calls itself n times is linear, provided other operations within the function don't depend on input size.
- A search algorithm that partitions the input into two parts and discards one at each iteration is logarithmic.

Time complexity in Python

Operation	list	dict(or set)	deque
Append	O(1)	–	O(1)
Insert	O(n)	O(1)	O(n)
Get item	O(1)	O(1)	O(n)
Set item	O(1)	O(1)	O(n)
Delete item	O(n)	O(1)	O(n)
x in S	O(n)	O(1)	O(n)
pop	O(1)	—	O(1)
pop(0)	O(n)	—	O(1)

Exercise: Which is the most appropriate data structure for each scenario and why?

1. A fixed collection of 100 integers.
2. A stack (first in, last out) of customer records.
3. A queue (first in, first out) of customer records.
4. A count of word occurrences within a document.

Complexity of algorithms

Complexity of certain important algorithms?

- Fast Fourier Transform: $O(n \log n)$
- Multiply two n-digit numbers using Karatsuba algorithm: $O(n^{1.59})$
- Matrix multiplication due to Coppersmith and Winograd: $O(n^{2.496})$
- Prime recognition of an n-digit integer due to Adleman, Pomerance and Rumley: $n^{O(\log \log n)}$
- Gaussian elimination: $O(n^3)$

Exercise

1. What is the complexity of a recursive implementation of the Fibonacci series?
2. What about our square root algorithm?

NumPy basics: importing a module/package in Python

What is `NumPy`?

```
NumPy is the fundamental package for scientific computing in
 Python. It is a Python library that provides a
multidimensional array object, various derived objects (
such as masked arrays and matrices), and an assortment of
routines for fast operations on arrays, including
mathematical, logical, shape manipulation, sorting,
selecting, I/O, discrete Fourier transforms, basic linear
algebra, basic statistical operations, random simulation
and much more.
```

```python
import numpy as np

```

NymPy: array typing

In general, NumPy arrays are constructed from sequences (e.g. lists), nesting as necessary for the number of desired dimensions.

Some properties of arrays:

- Arrays have a fixed size at creation
- All data must be homogeneous (consistent type)
- Built to support vectorized operations
- Avoids copying whenever possible

Examples

```
np.array([1,2,3])
## array([1, 2, 3])

np.array([[1,2],[3,4]])
## array([[1, 2],
##        [3, 4]])

np.array([[[1,2],[3,4]], [[5,6],[7,8]]])

```

NumPy: `dtype`

`NumPy` arrays will have a specific type used for storing their data, called their `dtype`. This is accessible via the `.dtype` attribute and can be set at creation using the `dtype` argument.

```
np.array([1,1]).dtype
## dtype('int64')

np.array([1.1, 2.2]).dtype
## dtype('float64')

np.array([True, False]).dtype
## dtype('bool')
```

```
np.array([1,2,3], dtype = np.uint8)
## array([1, 2, 3], dtype=uint8)

np.array([1,2,1000], dtype = np.uint8)
## array([  1,   2, 232], dtype=uint8)

np.array([3.14159, 2.33333], dtype = np.double)
## array([3.14159, 2.33333])
```

See here for more detailed description.

NumPy: creating 1d arrays

Some common tools for creating useful 1d arrays:

```
np.arange(10)
## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(3, 5, 0.25)
## array([3.  , 3.25, 3.5 , 3.75, 4.  , 4.25, 4.5 , 4.75])

np.linspace(0, 1, 11)
## array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
np.ones(4)
## array([1., 1., 1., 1.])

np.zeros(6)
## array([0., 0., 0., 0., 0., 0.])

np.full(3, False)
## array([False, False, False])

np.empty(4)
## array([1., 1., 1., 1.]
```

For the full list of creation, functions see here.

NumPy: creating 2d arrays (matrices)

Many of the same functions exist with some additional useful tools for common matrices,

```
np.eye(3)
## array([[1., 0., 0.],
##        [0., 1., 0.],
##        [0., 0., 1.]])

np.identity(2)
## array([[1., 0.],
##        [0., 1.]])

np.zeros((2,2))
## array([[0., 0.],
##        [0., 0.]])
```

```
np.diag([3,2,1])
## array([[3, 0, 0],
##        [0, 2, 0],
##        [0, 0, 1]])

np.tri(3)
## array([[1., 0., 0.],
##        [1., 1., 0.],
##        [1., 1., 1.]])

np.triu(np.full((3,3),3))
```

np.matrix is no longer recommended; use the ndarray class instead.

NumPy: subsetting an array

Arrays are subsetted using the standard Python syntax with either indexes or slices. Commas separate different dimensions.

```
x = np.array([[1,2,3],[4,5,6],[7,8,9]])
x
## array([[1, 2, 3],
##        [4, 5, 6], [7, 8, 9]])

x[0]
## array([1, 2, 3])

x[0,0]
## 1

x[0][0]
## 1
x[0:3:2, :]
## array([[1, 2, 3], [7, 8, 9]])

```

NumPy: views and copies

Basic subsetting of ndarray objects does not result in a new object but instead a ”view” of the original object. There are a couple of ways that we can investigate this behaviour

```
x = np.arange(10)
y = x[2:5]
z = x[2:5].copy()
print("x =", x, ", x.base =", x.base)
## x = [0 1 2 3 4 5 6 7 8 9] , x.base = None

print("y =", y, ", y.base =", y.base)
## y = [2 3 4] , y.base = [0 1 2 3 4 5 6 7 8 9]

print("z =", z, ", z.base =", z.base)
## z = [2 3 4] , z.base = None

type(x); type(y); type(z)
## <class 'numpy.ndarray'>
## <class 'numpy.ndarray'>
## <class 'numpy.ndarray'>
```

NumPy: subsetting with ...

There is some special syntax available using ... which expands to the number of: needed to account for all dimensions,

```
x = np.arange(16).reshape(2,2,2,2)
x[0, 1, ...]
## array([[4, 5],
##        [6, 7]])

x[..., 1]

x[0, 1, :, :]
## try and check the output
x[:, :, :, 1]
```

NumPy: subsetting with tuples or list

Unlike lists, a ndarray can be subset by a tuple containing integers

```
x = np.arange(6)
x
## array([0, 1, 2, 3, 4, 5])

x[(0,1,3),]

## array([0, 1, 3])

x[(0,1,3)]

## Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: too many indices for array: array is 1-
dimensional, but 3 were indexed
```

Exercise

Check NumPy's documentation to understand why the previous error occurs.

NumPy: subsetting assignment

Most of the subsetting approaches we've just seen can also be used for assignments. Just remember that we cannot change the size or type of the ndarray.

```
x = np.arange(9).reshape((3,3)); x
## array([[0, 1, 2],
##        [3, 4, 5],
##        [6, 7, 8]])

x[0,0] = -1
x
## array([[-1,  1,  2],
##        [ 3,  4,  5],
##        [ 6,  7,  8]])

x[0, :] = -2
x
## array([[-2, -2, -2],
##        [ 3,  4,  5],
##        [ 6,  7,  8]])

```

NumPy: subsetting assignment

More examples....

```
x[0:2,1:3] = -3
x
## array([[-2, -3, -3],
##        [ 3, -3, -3],
##        [ 6,  7,  8]])

x[(0,1,2), (0,1,2)] = -4
x
## array([[-4, -3, -3],
##        [ 3, -4, -3],
##        [ 6,  7, -4]])

```

NumPy: reshaping arrays

The dimensions of an array can be retrieved via the .shape attribute, and these values can be changed using the followings:

```
x = np.arange(6)
x
## array([0, 1, 2, 3, 4, 5])

y = x.reshape((2,3))
y
## array([[0, 1, 2],
##        [3, 4, 5]])

np.shares_memory(x,y)
## True
```

```
z = x
z.shape = (2,3)
z
## array([[0, 1, 2],
##        [3, 4, 5]])

x
## array([[0, 1, 2],
##        [3, 4, 5]])

np.shares_memory(x,z)
## True
```

Check the NumPy doc for more examples and exercises.

NumPy: implicit dimensions

When reshaping an array, the value -1 can be used to calculate a dimension automatically.

```
x = np.arange(6)
## array([0, 1, 2, 3, 4, 5])

x.reshape((2,-1))
## array([[0, 1, 2],
##        [3, 4, 5]])

x.reshape((-1,3,2))
## array([[[0, 1],
##         [2, 3],
##         [4, 5]]])

x.reshape(-1)
## array([0, 1, 2, 3, 4, 5])

x.reshape((-1,4))
## output = ??

```

NumPy: flattening arrays

We have just seen the most common approach to flattening an array (.reshape(-1)). There are two additional methods/functions:

1. `ravel` creates a flattened view of the array
2. `flatten` creates a flattened copy of the array

```
x = np.arange(6).reshape((2,3))
## array([[0, 1, 2],
##        [3, 4, 5]])

y = x.ravel()
## array([0, 1, 2, 3, 4, 5])

np.shares_memory(x,y)
## True

z = x.flatten()
np.shares_memory(x,z)
## False

```

NumPy: joining arrays

`concatenate()` is a general purpose function for this, with specialized versions `hstack()`, `vstack()`, and `dstack()`.

```
x = np.arange(4).reshape((2,2));
## array([[0, 1],[2, 3]])

y = np.arange(4,8).reshape((2,2));
## array([[4, 5],[6, 7]])

np.concatenate((x,y), axis=0)
## array([[0, 1],[2, 3],
##        [4, 5], [6, 7]])

np.concatenate((x,y), axis=1)
## array([[0, 1, 4, 5], [2, 3, 6, 7]])
## What about concatenate((x,y), axis=2) ?

np.concatenate((x,y), axis=None)
## array([0, 1, 2, 3, 4, 5, 6, 7]), try np.vstack()
```