Geometric & Graphics Programming Lab: Lecture 2

Alberto Paoluzzi

October 7, 2016

1. Python tutorials
2. IPython Notebook (Jupyter)
3. Assignments
4. Bootstrap `pyplasm` and `larlib`
5. Workshops: number 1

Python tutorials

Python tutorial links

1. Learn X in Y minutes where X = Python

Python tutorial links

1. Learn X in Y minutes where X = Python
2. Rudimenti-di-python

Python tutorial links

1. Learn X in Y minutes where X = Python
2. Rudimenti-di-python
3. Intro to python

1. Primitive Datatypes and Operators

From [Learn Python in Y minutes](https://learnxinyminutes.com/docs/python/)

```
# You have numbers
3  # => 3

# Math is what you would expect
1 + 1  # => 2
8 - 1  # => 7
10 * 2  # => 20
35 / 5  # => 7

# Division is a bit tricky. It is integer division and floors the results
# automatically.
5 / 2  # => 2

# To fix division we need to learn about floats.
2.0     # This is a float
11.0 / 4.0  # => 2.75 ahhh...much better

# Result of integer division truncated down both for positive and negative.
5 // 3     # => 1
5.0 // 3.0 # => 1.0 # works on floats too
-5 // 3  # => -2
-5.0 // 3.0 # => -2.0
```

Some number operations

```
# Note that we can also import division module(Section 6 Modules)
# to carry out normal division with just one '/'.
from __future__ import division
11/4    # => 2.75  ...normal division
11//4   # => 2 ...floored division

# Modulo operation
7 % 3 # => 1

# Exponentiation (x to the yth power)
2**4 # => 16

# Enforce precedence with parentheses
(1 + 3) * 2  # => 8
```

Boolean Operators

```
# Note "and" and "or" are case-sensitive
True and False #=> False
False or True #=> True

# Note using Bool operators with ints
0 and 2 #=> 0
-5 or 0 #=> -5
0 == False #=> True
2 == True #=> False
1 == True #=> True

# negate with not
not True  # => False
not False  # => True
```

Comparison operators

```
# Equality is ==
1 == 1  # => True
2 == 1  # => False

# Inequality is !=
1 != 1  # => False
2 != 1  # => True

# More comparisons
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# Comparisons can be chained!
1 < 2 < 3  # => True
2 < 3 < 2  # => False
```

Strings

```
# Strings are created with " or '
"This is a string."
'This is also a string.'

# Strings can be added too!
"Hello " + "world!"  # => "Hello world!"
# Strings can be added without using '+'
"Hello " "world!"  # => "Hello world!"

# ... or multiplied
"Hello" * 3  # => "HelloHelloHello"

# A string can be treated like a list of characters
"This is a string"[0]  # => 'T'

# You can find the length of a string
len("This is a string")  # => 16
```

String formatting

```
#String formatting with %
#Even though the % string operator will be deprecated on Python 3.1 and removed
#later at some time, it may still be good to know how it works.
x = 'apple'
y = 'lemon'
z = "The items in the basket are %s and %s" % (x,y)

# A newer way to format strings is the format method.
# This method is the preferred way
"{} is a {}".format("This", "placeholder")
"{0} can be {1}".format("strings", "formatted")
# You can use keywords if you don't want to count.
"{name} wants to eat {food}".format(name="Bob", food="lasagna")
```

2. Variables and Collections

From [Learn Python in Y minutes](https://learnxinyminutes.com/docs/python/)

```
# Python has a print statement
print "I'm Python. Nice to meet you!" # => I'm Python. Nice to meet you!

# Simple way to get input data from console
input_string_var = raw_input("Enter some data: ") # Returns the data as a string
input_var = input("Enter some data: ") # Evaluates the data as python code
# Warning: Caution is recommended for input() method usage
# Note: In python 3, input() is deprecated and raw_input() is renamed to input()

# No need to declare variables before assigning to them.
some_var = 5    # Convention is to use lower_case_with_underscores
some_var  # => 5

# Accessing a previously unassigned variable is an exception.
# See Control Flow to learn more about exception handling.
some_other_var  # Raises a name error

# if can be used as an expression
# Equivalent of C's '?:' ternary operator
"yahoo!" if 3 > 2 else 2  # => "yahoo!"
```

Lists

```
# Lists store sequences
li = []
# You can start with a prefilled list
other_li = [4, 5, 6]

# Add stuff to the end of a list with append
li.append(1)    # li is now [1]
li.append(2)    # li is now [1, 2]
li.append(4)    # li is now [1, 2, 4]
li.append(3)    # li is now [1, 2, 4, 3]
# Remove from the end with pop
li.pop()        # => 3 and li is now [1, 2, 4]
# Let's put it back
li.append(3)    # li is now [1, 2, 4, 3] again.

# Access a list like you would any array
li[0]  # => 1
# Assign new values to indexes that have already been initialized with =
li[0] = 42
li[0]  # => 42
li[0] = 1  # Note: setting it back to the original value
# Look at the last element
li[-1]  # => 3
```

Lists

```
# Looking out of bounds is an IndexError
li[4]  # Raises an IndexError

# You can look at ranges with slice syntax.
# (It's a closed/open range for you mathy types.)
li[1:3]  # => [2, 4]
# Omit the beginning
li[2:]  # => [4, 3]
# Omit the end
li[:3]  # => [1, 2, 4]
# Select every second entry
li[::2]   # =>[1, 4]
# Reverse a copy of the list
li[::-1]   # => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]
```

Lists

```
# Remove arbitrary elements from a list with "del"
del li[2]   # li is now [1, 2, 3]

# You can add lists
li + other_li   # => [1, 2, 3, 4, 5, 6]
# Note: values for li and for other_li are not modified.

# Concatenate lists with "extend()"
li.extend(other_li)   # Now li is [1, 2, 3, 4, 5, 6]

# Remove first occurrence of a value
li.remove(2)  # li is now [1, 3, 4, 5, 6]
li.remove(2)  # Raises a ValueError as 2 is not in the list

# Insert an element at a specific index
li.insert(1, 2)  # li is now [1, 2, 3, 4, 5, 6] again

# Get the index of the first item found
li.index(2)  # => 1
li.index(7)  # Raises a ValueError as 7 is not in the list

# Check for existence in a list with "in"
1 in li   # => True
```

Tuples

```
# Tuples are like lists but are immutable.
tup = (1, 2, 3)
tup[0]   # => 1
tup[0] = 3  # Raises a TypeError

# You can do all those list thingies on tuples too
len(tup)   # => 3
tup + (4, 5, 6)   # => (1, 2, 3, 4, 5, 6)
tup[:2]   # => (1, 2)
2 in tup   # => True

# You can unpack tuples (or lists) into variables
a, b, c = (1, 2, 3)     # a is now 1, b is now 2 and c is now 3
d, e, f = 4, 5, 6       # you can leave out the parentheses
# Tuples are created by default if you leave out the parentheses
g = 4, 5, 6             # => (4, 5, 6)
# Now look how easy it is to swap two values
e, d = d, e     # d is now 5 and e is now 4
```

Dictionaries

```
# Dictionaries store mappings
empty_dict = {}
# Here is a prefilled dictionary
filled_dict = {"one": 1, "two": 2, "three": 3}

# Look up values with []
filled_dict["one"]   # => 1

# Get all keys as a list with "keys()"
filled_dict.keys()   # => ["three", "two", "one"]
# Note - Dictionary key ordering is not guaranteed.
# Your results might not match this exactly.

# Get all values as a list with "values()"
filled_dict.values()   # => [3, 2, 1]
# Note - Same as above regarding key ordering.

# Get all key-value pairs as a list of tuples with "items()"
filled_dicts.items()    # => [("one", 1), ("two", 2), ("three", 3)]
```

Dictionaries

```
# Check for existence of keys in a dictionary with "in"
"one" in filled_dict   # => True
1 in filled_dict   # => False

# Looking up a non-existing key is a KeyError
filled_dict["four"]   # KeyError

# Use "get()" method to avoid the KeyError
filled_dict.get("one")   # => 1
filled_dict.get("four")   # => None
# The get method supports a default argument when the value is missing
filled_dict.get("one", 4)   # => 1
filled_dict.get("four", 4)   # => 4
# note that filled_dict.get("four") is still => None
# (get doesn't set the value in the dictionary)

# set the value of a key with a syntax similar to lists
filled_dict["four"] = 4  # now, filled_dict["four"] => 4

# "setdefault()" inserts into a dictionary only if the given key isn't present
filled_dict.setdefault("five", 5)  # filled_dict["five"] is set to 5
filled_dict.setdefault("five", 6)  # filled_dict["five"] is still 5
```

Sets

```
# Sets store ... well sets (which are like lists but can contain no duplicates)
empty_set = set()
# Initialize a "set()" with a bunch of values
some_set = set([1, 2, 2, 3, 4])   # some_set is now set([1, 2, 3, 4])

# order is not guaranteed, even though it may sometimes look sorted
another_set = set([4, 3, 2, 2, 1])  # another_set is now set([1, 2, 3, 4])

# Since Python 2.7, {} can be used to declare a set
filled_set = {1, 2, 2, 3, 4}   # => {1, 2, 3, 4}

# Add more items to a set
filled_set.add(5)   # filled_set is now {1, 2, 3, 4, 5}
```

Sets

```
# Do set intersection with &
other_set = {3, 4, 5, 6}
filled_set & other_set   # => {3, 4, 5}

# Do set union with |
filled_set | other_set   # => {1, 2, 3, 4, 5, 6}

# Do set difference with -
{1, 2, 3, 4} - {2, 3, 5}   # => {1, 4}

# Do set symmetric difference with ^
{1, 2, 3, 4} ^ {2, 3, 5}  # => {1, 4, 5}

# Check if set on the left is a superset of set on the right
{1, 2} >= {1, 2, 3} # => False

# Check if set on the left is a subset of set on the right
{1, 2} <= {1, 2, 3} # => True

# Check for existence in a set with in
2 in filled_set   # => True
10 in filled_set   # => False
```

3. Control Flow

{From Learn Python in Y minutes}

If statement

```
# Let's just make a variable
some_var = 5


# Here is an if statement. Indentation is significant in python!
# prints "some_var is smaller than 10"

if some_var > 10:
    print "some_var is totally bigger than 10."
elif some_var < 10:    # This elif clause is optional.
    print "some_var is smaller than 10."
else:              # This is optional too.
    print "some_var is indeed 10."
```

For loops

```
"""
For loops iterate over lists
prints:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    # You can use {0} to interpolate formatted strings. (See above.)
    print "{0} is a mammal".format(animal)
```

range(number)

```
"""
"range(number)" returns a list of numbers
from zero to the given number
prints:
    0
    1
    2
    3
"""
for i in range(4):
    print i
```

range(lower, upper)

```
"""
"range(lower, upper)" returns a list of numbers
from the lower number to the upper number
prints:
    4
    5
    6
    7
"""
for i in range(4, 8):
    print i
```

While loops

```
"""
While loops go until a condition is no longer met.
prints:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print x
    x += 1  # Shorthand for x = x + 1
```

Exceptions

```
# Handle exceptions with a try/except block

# Works on Python 2.6 and up:
try:
    # Use "raise" to raise an error
    raise IndexError("This is an index error")
except IndexError as e:
    pass    # Pass is just a no-op. Usually you would do recovery here.
except (TypeError, NameError):
    pass    # Multiple exceptions can be handled together, if required.
else:   # Optional clause to the try/except block. Must follow all except blocks
    print "All good!"   # Runs only if the code in try raises no exceptions
finally: #  Execute under all circumstances
    print "We can clean up resources here"
```

with statement

```
# Instead of try/finally to cleanup resources you can use a with statement
with open("myfile.txt") as f:
    for line in f:
        print line
```

4. Functions

From [Learn Python in Y minutes](https://learnxinyminutes.com/docs/python/)

```
# Use "def" to create new functions
def add(x, y):
    print "x is {0} and y is {1}".format(x, y)
    return x + y    # Return values with a return statement

# Calling functions with parameters
add(5, 6)   # => prints out "x is 5 and y is 6" and returns 11

# Another way to call functions is with keyword arguments
add(y=6, x=5)   # Keyword arguments can arrive in any order.


# You can define functions that take a variable number of
# positional args, which will be interpreted as a tuple by using *
def varargs(*args):
    return args

varargs(1, 2, 3)   # => (1, 2, 3)
```

keyword args

```
# You can define functions that take a variable number of
# keyword args, as well, which will be interpreted as a dict by using **
def keyword_args(**kwargs):
    return kwargs

# Let's call it to see what happens
keyword_args(big="foot", loch="ness")    # => {"big": "foot", "loch": "ness"}


# You can do both at once, if you like
def all_the_args(*args, **kwargs):
    print args
    print kwargs
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""
```

Expand positional/keyword args

```
# When calling functions, you can do the opposite of args/kwargs!
# Use * to expand positional args and use ** to expand keyword args.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)   # equivalent to foo(1, 2, 3, 4)
all_the_args(**kwargs)   # equivalent to foo(a=3, b=4)
all_the_args(*args, **kwargs)   # equivalent to foo(1, 2, 3, 4, a=3, b=4)
```

Pass args and kwargs

```
# you can pass args and kwargs along to other functions that take args/kwargs
# by expanding them with * and ** respectively
def pass_all_the_args(*args, **kwargs):
    all_the_args(*args, **kwargs)
    print varargs(*args)
    print keyword_args(**kwargs)
```

Function Scope

```
x = 5

def set_x(num):
    # Local var x not the same as global variable x
    x = num # => 43
    print x # => 43

def set_global_x(num):
    global x
    print x # => 5
    x = num # global var x is now set to 6
    print x # => 6

set_x(43)
set_global_x(6)
```

First class and anonymous functions

```
# Python has first class functions
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)   # => 13

# There are also anonymous functions
(lambda x: x > 2)(3)    # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1) # => 5
```

built-in higher order functions

```
# There are built-in higher order functions
map(add_10, [1, 2, 3])   # => [11, 12, 13]
map(max, [1, 2, 3], [4, 2, 1])   # => [4, 2, 3]

filter(lambda x: x > 5, [3, 4, 5, 6, 7])   # => [6, 7]
```

List comprehensions

```
# We can use list comprehensions for nice maps and filters
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]   # => [6, 7]
```

Use it: is compiled efficiently!

Set and dict comprehensions

```
# You can construct set and dict comprehensions as well.
{x for x in 'abcddeef' if x in 'abc'}  # => {'d', 'e', 'f'}
{x: x**2 for x in range(5)}  # => {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

5. Classes

From [Learn Python in Y minutes](https://learnxinyminutes.com/docs/python/)

6. Modules

From [Learn Python in Y minutes](https://learnxinyminutes.com/docs/python/)

7. Advanced

From [Learn Python in Y minutes](https://learnxinyminutes.com/docs/python/)

IPython Notebook (Jupyter)

Jupyter

The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, $equations$, visualizations and explanatory text.

Remark 1

Use Jupiter for your lab works

Remark 2

And store the files in GitHub at your account

Assignments

Assignments

1. create YOUR ggpl GitHub repository: NOW !!

Assignments

1. create YOUR ggpl GitHub repository: NOW !!

2. send me an email with link to it
   please use [ggpl] markup on email "Subject": NOW !!

Assignments

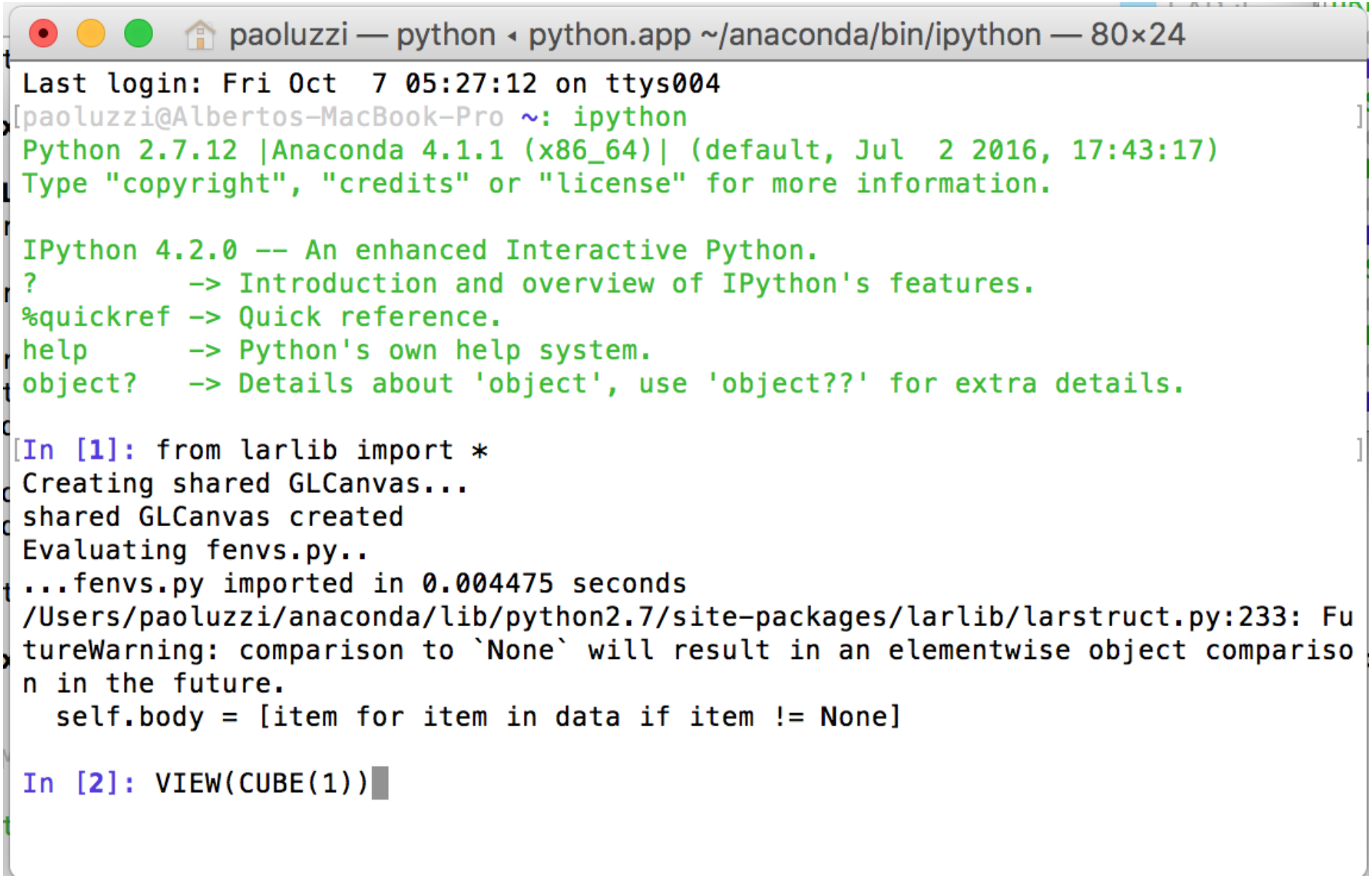
1. create YOUR ggpl GitHub repository: NOW !!

2. send me an email with link to it
   please use [ggpl] markup on email "Subject": NOW !!

3. Download and/or install your Jupyter Notebook: NOW !!

Bootstrap pyplasm and larlib

Open IPython and import larlib



Workshops: number 1

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings
- using few `pyplasm` primitive operations

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings
- using few `pyplasm` primitive operations
- parameterized by:

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings
- using few `pyplasm` primitive operations
- parameterized by:
  - `(bx,bz)` (given dimensions of beam section)

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings
- using few `pyplasm` primitive operations
- parameterized by:
  - `(bx,bz)` (given dimensions of beam section)
  - `(px,py)` (given dimensions of pillar section)

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings
- using few `pyplasm` primitive operations
- parameterized by:
  - `(bx,bz)` (given dimensions of beam section)
  - `(px,py)` (given dimensions of pillar section)
  - `[dy1,dy2,...]` (distances between axes of the pillars)

Simple parametric building structure in reinforced concrete

- Space frame of reinforced concrete
- including beams, pillars and footings
- using few `pyplasm` primitive operations
- parameterized by:
  - `(bx,bz)` (given dimensions of beam section)
  - `(px,py)` (given dimensions of pillar section)
  - `[dy1,dy2,...]` (distances between axes of the pillars)
  - `[hz1,hz2,...]` (interstory heights)