Python and MILP Cheatsheet

Pablo Angulo for ESTIN@UPM.ES

python

Base types

`bool` *booleans* takes only two values: `True` and `False`.

`int` positive and negative *integers*, not bounded

`float` *floating point* numbers approximate any real number (e.g. `-1.25e-6` ($-1.26 \cdot 10^{-6}$))

`str` A *string* is a sequence of characters. Always represented with quotes or double quotes (e.g. `'Hello, World'`, or `"Hello, World"`)

Variables

Identifiers start with a letter or _, may contain numbers.
`identifier = value` binds the value to the identifier

`my_number = 1` Binds identifier `my_number` to the integer 1.

`my_number += 10` Equivalent to `my_number = my_number + 10`.

`a=b=1` Binds both identifiers `a` and `b` to the integer 1.

`a,b = 1,2` Binds `a` to 1, and `b` to 2.

`a,b = ["one", "two"]` *unpacks* list `["one", "two"]`, binding `a` to `"one"`, and `b` to `"two"`

Inmutable container types

Inmutable types can not be modified, but new objects can be built from the old ones.

`str` Can only hold characters (e.g. `a = "Hello, World"`).

`tuple` May hold any data type (e.g. `b = (12, True, "abc")`).

Operations with containers

`len(container)` Returns the number of elements in `container`.

`container[index]` Access an element within the container. Index starts at 0, last element has index -1: (e.g. `b[0]`⇒ 12, `print(b[-1])`⇒ `"abc"`).

`container[start:end]` Get a subsequence, or slice. `start` index is included, `end` index is excluded: (e.g. `a[0:5]`⇒ `"Hello"`).

`container1 + container2` concatenate compatible containers (e.g. `a + "!"`⇒ `"Hello, World!"`).

`container*number` repeat container (e.g. `"abc"*3`⇒ `"abcabcabc"`).

Mutable container types

`dict` Holds (key, value) pairs (e.g. `d = {1:"one", 2:"two"}`).

`d[1]="uno"` Update an existing element `d` ⇒ `{1:"uno", 2:"two"}`).

`d[4]="four"` Adds a new (key,value) pair `d` ⇒ `{1:"uno", 2:"two", 4:"four"}`).

`list` May hold any data type (e.g. `l = [12, True, "abc"]`).

`l[1:3]=[False, False]` Updates a whole slice `l` ⇒ `[12, False, False]`).

`l.append(1e3)` Add an element at the end of the list `l` ⇒ `[12, False, False, 1e3]`).

Conversions

`int(5.46)` `int( )` of a float truncates the decimal part.

`int("14")` returns the integer `14`.

`float("12.3")` returns the float `12.3`.

`str(2.34)` returns the string `"2.34"`.

`bool(x)` return `False` if `x` is `None`, the boolean `False`, the number `0`, or an empty container.

Conditionals

Exactly one of the indented blocks of an "`if/elif/else`" statement will be executed (or maybe none of them if there is no "`else`" clause):

```
if some_condition:
    do_this()
elif other_condition:
    do_that()
else:
    do_whatever()
```

Combine conditions with `and`, `or`, and using parenthesis.

```
# Lines that start with # are comments, and are not executed
if (a>0) and (a+b<=1):
    do_this()
    and_this()
# One equal sign = for assignment, two of them == for comparison
# != for "not equal"
elif (len(d)!=1) or (a=="Hello, Planet"):
    do_that()

# next line is not indented, so it will be executed in any case
and_do_this_regardless()
```

Loops

`for` loops repeat some statements while one variable runs over the elements of a container:

```
s = 0
# when the loop starts, s is 0
for x in [1,2,3,4]:
    s = s + x
# when the loop ends, s is 1+2+3+4 = 10
# the final value, 10, will be printed only once
print(s)
```

`while` loops repeat some statements while a certain condition is satisfied

```
s = 0
# when the loop starts, s is 0
while s < 5:
    s = s + 1
# when the loop ends, s is 5
# the final value, 5, will be printed only once
print(s)
```

Functions

In the **function definition**, the **body** of the function is indented. Statements in the function body are not executed when the function is defined.

```
def suma(x,y):
    '''this text describes the purpose of the function'''
    s = x + y
    return s
```

In a **function call**, the body of the function is executed.

```
#the identifier z is bound to the integer 3
z = suma(1,2)
```

python libraries

import modules

- Import a module

  ```
  import numpy
  my_array = numpy.zeros(10)
  ```

- Import a module using an alias

  ```
  import numpy as np
  x = np.pi/2
  ```

- Import specific functions from a module

  ```
  from numpy import sin, arcsin
  # prints "1"
  print( sin(arcsin(1)) )
  ```

numpy

`numpy` provides `arrays`, which are mutable data structures, but with fixed size. They are very efficient, and are designed for numerical computation. Many famous libraries are built on top of `numpy`.

```
import numpy as np
```

- Build an array from a list

  `xs = np.array([1,10,100])`

- Add a number to all elements of the array

  `xs + 1`

  ⇒ `array([1,11,101])`

- Apply a function to all elements of the array

  `np.log10(xs)`

  ⇒ `array([0,1,2])`

- Add two arrays

  `xs + np.log10(xs)`

  ⇒ `array([1,11,102])`

- Fill a one dimensional array with numbers from 0 to n (n is not included)

  `np.arange(3)`

  ⇒ `array([0, 1, 2])`

- Fill a one dimensional array with n floating points equispaced from a to b (both a and b are included)

  `np.linspace(1,2,5)`

  ⇒ `array([1, 1.25, 1.5, 1.75, 2])`

- Fill a one dimensional array with `n` zeros

  `zs = np.zeros(n)`

- Fill a two dimensional, $n \times m$ array (a.k.a. a matrix) with zeros.

  `A = np.zeros((n, m))`

- Fill a 5×5 array with the value 7.

  `B = 7*np.ones((5,5))`

- `*` is the element-wise product of arrays, `@` is the matrix "dot product".

  ```
  B = 7*np.ones((5,5))
  # 5x5 identity matrix
  Id = np.eye(5)
  # D is diagonal, with '7' in the diagonal
  D = Id*B
  # E = B
  E = Id@B
  ```

matplotlib

`matplotlib` can build many types of graphics that represent quantitative information. The submodule `pyplot` makes it easy to use.

```
import matplotlib.pyplot as plt

# 101 points: 0, 0.01, 0.02... 0.99, 1
xs = np.linspace(0, 1, 101)
# sine and cosine are evaluated at each of those points
fs = np.sin(2*np.pi*xs)
gs = np.cos(2*np.pi*xs)

plt.figure(figsize=(10,5))
# the graph of the sine is a dotted blue line
# the graph of the cosine is a solid green line
plt.plot(xs, fs, "b.", label="graph of f")
plt.plot(xs, gs, "g-", label="graph of g")
plt.legend()
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.title("A graph that combines two plots")
```

Linear programming

optimization problems

Decision variables. Some variables $x_1, \ldots, x_n$ whose value we can choose.

Constraints. The decision variables must satisfy *all* the constraints:

Equality constraint: $g(x_1, \ldots, x_n) = c$, for a certain function g of the decision variables.

Inequality constraint: $h(x_1, \ldots, x_n) \leq b$, for a certain function h of the decision variables.

Objective. A function $f(x_1, \ldots, x_n)$ that we want to either minimize (e.g. the less cost, the better) or maximize (e.g. the more welfare, the better).

LP and MILP problems

An optimization problem where the objective function, and all the constraints, are linear functions:

- In a **Linear Programming (LP)** problem, all decision variables can take any real value, provided that all the constraints are satisfied.
- In a **Mixed Integer Linear Programming (MILP)** problem, some decision variables can take any real value (*continuous variables*), but others may only take integer values (*integer variables*).

The relaxed problem

For a given *MILP* problem, the **relaxed problem** is the *LP* program with the same decision variables, objective and constraints, but the requirement that some variables must be integer is removed.

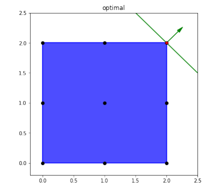
Feasible region

If there are n decision variables $x_1, \ldots, x_n$, the **feasible region** is the subset of $\mathbb{R}^n$ consisting of the points of $\mathbb{R}^n$ that satisfy all the constraints.
A feasible region can be **bounded**, **unbounded** or **empty**.

Classification of MILP problems

Unique optimal solution

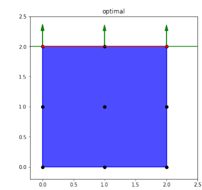
Optimal value is attained at exactly one point of the feasible region.



$$\begin{array}{rr} \text{Max:} & x+y \\ \text{Such that:} & 0 \leq x \leq 2 \\ & 0 \leq y \leq 2 \\ & x, y \in \mathbb{Z} \end{array}$$

Multiple optimal solution

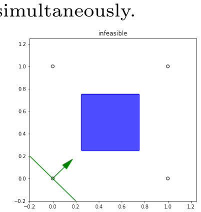
Optimal value is attained at more than one point of the feasible region.



$$\begin{array}{rr} \text{Max:} & y \\ \text{Such that:} & 0 \leq x \leq 2 \\ & 0 \leq y \leq 2 \\ & x, y \in \mathbb{Z} \end{array}$$

Infeasible

Feasible region is empty: it is impossible to satisfy all the constraints simultaneously.

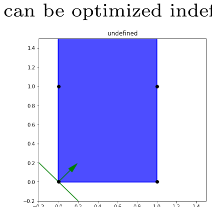


$$\begin{array}{rr} \text{Max:} & x+y \\ \text{Such that:} & 1 \leq 4x \leq 3 \\ & 1 \leq 4y \leq 3 \\ & x, y \in \mathbb{Z} \end{array}$$

(the feasible region of the *relaxed* problem is shown in blue)

Unbounded

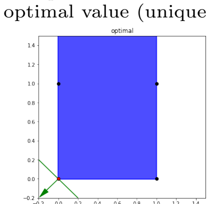
Feasible region is unbounded, and furthermore, the objective function can be optimized indefinitely.



$$\begin{array}{rr} \text{Max:} & x+y \\ \text{Such that:} & 0 \leq x \leq 1 \\ & 0 \leq y \\ & x, y \in \mathbb{Z} \end{array}$$

Unbounded region, but with optimal value

It is possible that the feasible region is unbounded, but there is a finite optimal value (unique or multiple):



$$\begin{array}{rr} \text{Min:} & x+y \\ \text{Such that:} & 0 \leq x \leq 1 \\ & 0 \leq y \\ & x, y \in \mathbb{Z} \end{array}$$

The Backpack Problem

Choose several items from a list, that fit into our backpack, and so that their total value is maximized

Item	Weight	Value
I_1	w_1	v_1
...	...	...
I_n	w_n	v_n

Decision variables

$$x_j = \begin{cases} 1 & \text{put item } j \text{ in backpack} \\ 0 & \text{do not put item } j \text{ in backpack} \end{cases}$$

Objective

Maximize total value, only items in the backpack contribute:

$$\text{Max: } \sum_{j=1}^{n} x_j v_j$$

Constraints

All x_j are either 0 or 1, and they fit in the backpack:

$$\begin{array}{rcl} \sum_{j=1}^{n} x_j w_j & \leq & \text{backpack capacity} \\ 0 & \leq & x_j, \text{ for } j = 1, \ldots, n \\ x_j & \leq & 1, \text{ for } j = 1, \ldots, n \\ x_j & \in & \mathbb{Z}, \text{ for } j = 1, \ldots, n \end{array}$$

optlang

```
from optlang import Model, Variable, Constraint,
    Objective

model = Model(name='optlang model')

### Decision variables, positive (lb is lower bound)
#   x is real, y is interger
x = Variable('x',lb=0,type='continuous')
y = Variable('y',lb=0,type='integer')

### Constraints, x+2*y<=4, 5*x-y>=8
model.add([
    Constraint(x+2*y, ub=4),
    Constraint(5*x-y, lb=8)
])

### Objetive function to be maximized
model.objective = Objective(x+2*y-2, direction='max')

### Solve
status = model.optimize()

### status can be "optimal", "infeasible", "unbounded"
#   or "undefined", if the solver decides there is no
#   optimal value, but cannot decide why
print("status:", model.status)

### optimal value
#   (only acceptable if status is "optimal")
print("objective value:", model.objective.value)
### print the value of each decision variable
#   for the optimal solution
#   (only acceptable if status is "optimal")
for var_name, var in model.variables.iteritems():
    print(var_name, "=", var.primal)
```