itom cheat sheet
Python 3

Help

help(m)	Display help for module, function…
pluginHelp("name")[1]	Display information for plugin
filterHelp("name")[1]	Display information for itom-filter
widgetHelp("name")[1]	Display information for widget in plugin
dir(m)	Display names in module *m*

Common Data Types

int	Integer (32/64bit)	3, -4, 0
float	Floating point number	3.0, -6.55, float('nan')
complex	Complex number	2+3j, 4j, 5-0j
bool	Boolean	True, False
str	String of characters	"Python"
byte	Sequence of integers	b"Python"
tuple	Immutable sequence	(2,), (2.3,"a"), (2.3,-1)
list	Mutable sequence	[2], [2.3,"a"], [2.3,-1]
dict	Mapping, dictionary	{"x":-2, "name":"a"}
numpy.array	Numpy-Array	
dataObject[1]	itom data object	compatible to np.array

Module Import

Example: How to call method *plot* of module *itom*

import itom	itom.plot(args,…)
from itom **import** plot	plot(args,…)
from itom **import** *	plot(args,…) [Import all]
from itom **import** plot **as** fct	fct(args,…) [Alias]

Operators and their Precedence

func_name(args,kwds)	Function call
x[startIdx : endIdx]	Slicing (startIdx incl., endIdx excl.)
x[index]	Indexing (index zero-based)
x.attribute	Attribute reference
**	Exponentation
*, /, %	Multiply, Divide, Mod
+, -	Add, Subtract
&, |, ^, ~	Binary And, Or, Xor, Not
>, <, <=, >=, !=, ==	Comparison
in, not in	Membership tests [2 in (1,2,3)]->True
not, and, or	Boolean operators

[1] only available in itom

Common Syntax Structures

exp [any expression], stmt [(sequence of) command(s)]
Note: Indentation is important for control sequences!

Assignment	a = 1 a, b = 1, 2 c = [1,2,3]; c[1] = 4	a=1 a=1,b=2 c=[1,4,3]
Output	print(exp [,expr2…])	print("test")
Comment	#single line	'''multi line'''

Selection	**if**(boolean_exp):
	stmt
	[**elif** (boolean_exp):
	stmt]
	[**else**:
	stmt]
		if(2>1):
		print("2>1")
		else:
		print("what?")

Repetition	**while**(boolean_exp): stmt	repeat while bool_exp is True
Traversal	**for** var **in** obj: stmt	Iterate over all elements in traversable obj.
Loop	**for** i **in range**(0,5): print(i)	Use range for creating an iterable list [0,1,2,3,4]
Exception Handling	**try**: stmt … **except** [exc_type] [,var]: stmt …	try: 1/0 except ZeroDivisionError: print("uups")
Function Definition	**def** fctname(params): '''doc-string''' stmt return obj	def test(i,j=4): a=i+j #j has default 4 return [a,"done"]
Function Call	ret = fctname(args)	ret = test(2) #ret is [6,"done"]

Common Built-in Functions

abs(x)	Absolute value of x
float(x), int(x)	Convert x to float / int (if possible)
len(s)	Number of items in sequence (list, tuple,...)
str(obj)	String representation of obj
range(x,y)	A list [x, x+1, x+2, ..., y-1] (**y excluded**)
dict()	Empty dictionary
list()	Empty list
tuple()	Empty tuple

Common Functions of Module *math* (from math import *)

cos(x), sin(x), tan(x)	Cosine, sine, tangent of *x* radians
sqrt(x)	Positive square root of x
degrees(x), radians(x)	Convert from rad to deg, deg to rad
exp(x)	e ** x
floor(x)	Largest whole number <= x
pow(x,y)	x ** y
pi	Math constant π (15 sig figs)
e	Math constant e (15 sig figs)

Common List (L) and Tuple (T) Methods

LT[idx], LT[idx1:idx2]	get items or slice of items from list/tuple
LT.count(obj)	number of occurrences of *obj* in LT
LT.index(obj)	index of first occurrence of *obj* in LT; raises ValueError if does not occur
L[idx]=obj	assigns new value to index (list only)
L.append(obj)	Appends *obj* to end of list L
L.remove(obj)	Removes first occurrence of *obj* from *L*

Common List (L) or Tuple (T) Methods, (LT both)

LT[idx], LT[idx1:idx2]	get items or slice of items from list/tuple
LT.count(obj)	number of occurrences of *obj* in LT
LT.index(obj)	index of first occurrence of *obj* in LT; raises ValueError if does not occur
L[idx]=obj	assigns new value to index (list only)
L.append(obj)	Appends *obj* to end of list L
L.remove(obj)	Removes first occurrence of *obj* from *L*

Common Dictionary (D) Methods

D["key"]	returns value corresponding to key
D["key"] = obj	replaces/adds *obj* under given key
"key" in D	True if key exists in D, else False
D.clear()	clears dictionary
D.keys()	Returns list of D's keys
D.values()	Returns list of D's values

Formatting Numbers as Strings

Syntax: "%width.precision type" % expression

width (optional)	total width (+/-: right/left aligned)
precision (optional)	specified digits of float precision
type (required)	d (int), f (float), s (string), e (exp. Notation)
Examples:	"%6d" % 123 -> …123 "%04d" % 1 -> 0001 "%8.2f" % 456.789 -> ..456.79 "%8.2e" % 456.789 -> 4.57e+02

Working with dataIO-Devices (Grabber, AD-Converter…)[1]	
pluginHelp("name")	Prints information about plugin
dataIO("name",params)	Creates obj (instance) of device
obj.getParam("name")	Returns value of parameter
obj.setParam("name",val)	Sets parameter to *val*
obj.startDevice()	Starts device (camera…)
obj.stopDevice()	Stops device (camera…)
obj.acquire()	triggers image acquisition
obj.getVal(dObj)	after call, dataObject dObj references to last acquired image
obj.copyVal(dObj)	after call, dObj contains deep copy of last acquired image
obj.setAutoGrabbing(bool)	En-/Disables continuous grab for connected live views

Working with actuator-Devices (Motors, Stages…)[1] Position units are in mm	
pluginHelp("name")	Prints information about plugin
actuator("name",params)	Creates obj (instance) of device
obj.getParam("name")	Returns value of parameter
obj.setParam("name",val)	Sets parameter to *val*
obj.getPos(idx1[,idx2…])	Returns current position for all given axes indices (0-based)
obj.setPosRel(idx1,pos1,…)	Relatively moves axis *idx1* by *pos1*
obj.setPosAbs(idx1,pos1,…)	Moves axis *idx1* to *pos1*

Working with itom-Filters[1]	
filterHelp("name")	Lists all algorithms/filters containing *name* or detailed information about filter that matches *name*
ret=filter("name",param1,…)	Calls filter *name* with given parameters and returns tuple of output parameters (or None)

Plots[1]	
plot(dObj)	1D or 2D plot of dObj (depending on its size)
liveImage(dataIO-instance)	Live view of camera

Common DataObject[1] **and Numpy.Array Data Types**	
"uint8", "int8", "uint16", "int16", "uint32", "int32"	(Un-)Signed integer 8,16,32 bit
"float32", "float64"	Floating point numbers
"complex64", "complex128"	Complex values (64 = 2x32 bit)
"rgba32"	Color value (type itom.rgba)

Numpy.array (import numpy as np, np.array)**, DataObject**[1] **(**import itom, itom.dataObject**)**		
arr=np.ndarray([2,3],'uint8')	dObj=dataObject([2,3],'uint8')	create a randomly filled 2x3 array with type uint8
arr=np.array([[1,2,3],[4,5,6]])	dObj =dataObject([2,3],data=(1,2,3,4,5,6))	create the 2x3 array [1,2,3 ; 4,5,6]
arr=np.array(dObj, copy=False)	dObj =dataObject(arr)	convert np.array <-> dataObject
arr.ndim	dObj.dims	Returns number of dimensions (*here:* 2)
arr.shape	dObj.shape	Returns size tuple (*here:* [2,3])
arr.shape[0]	dObj.size(0) *or* dObj.shape[0]	Returns size of first dimensions (*here:* y-axis)
c=arr[0,1]; arr[0,1]=7	dObj [0,1]; b[0,1]=7	Gets or sets the element in the 1st row, 2nd col
c=arr[:,1:3] *or* c=arr[0:2,1:3]	c=dObj[:,1:3] *or* c= dObj [0:2,1:3]	Returns shallow copy of array containing the 2nd and 3rd columns
arr[:,:]=7	dObj [:,:]=7	sets all values of array to value 7
arr.transpose() (*shallow copy*)	dObj.trans() (*deep copy*)	transpose of array
np.dot(arr1,arr2)	dObj1 * dObj2 *(float only)*	matrix multiplication
arr1 * arr2	dObj1.mul(dObj2)	element-wise multiply
arr1 / arr2	dObj1.div(dObj2)	element-wise divide
arr1 +,- arr2	dObj1 +,- dObj2	sum/difference of elements
arr1 +,- scalar	dObj1 +,- scalar	adds/subtracts scalar from every element in array
arr1 &,| arr2	dObj1 &,| dObj2	element-wise, bitwise AND/OR operator
arr2 = arr1	dObj2 = dObj1	referencing (both still point to the same array)
arr2 = arr1.copy()	dObj2 = dObj1.copy()	deep copy (entire data is copied)
arr2 = arr1.astype(newtype)	dObj2 = dObj1.astype('newtypestring')	type conversion
arr = np.zeros([3,4],'float32')	dObj = dataObject.zeros([3,4], 'float32')	3x4 array filled with zeros of type float32
arr = np.ones([3,4],'float32')	dObj = dataObject.ones([3,4], 'float32')	3x4 array filled with ones of type float32
arr = np.eye(3,dtype='float32')	dObj = dataObject.eye(3, 'float32')	3x3 identity matrix (type: float32)
arr2 = arr1.squeeze()	dObj2 = dObj1.squeeze()	converts array to an array where dimensions of size 1 are eliminated (deep copy if necessary)
np.linspace(1,3,4)	-	4 equally spaced samples between 1 and 3, inclusive
[x,y] = np.meshgrid(0:2,1:5)	-	two 2D arrays: one of x values, the other of y values
np.linalg.inv(a)	-	inverse of square matrix *a*
x=np.linalg.solve(a,b)	-	solution of *ax=b* (using pseudo inverse)
[U,S,V] = np.linalg.svd(a)	-	singular value decomposition of *a* (V is transposed!)
np.fft.fft2(a), np.fft.ifft2(a)	filter available	(Inverse) 2D fourier transform of *a*
a[a>0]=5	a[a>0]=5	sets all elements > 0 of *a* to 5
a[np.isnan(a)]=5	a[np.isnan(a)]=5	sets all NaN elements of a to 5
arr2 = arr1.reshape([3,2])	arr2 = arr1.reshape([3,2])	reshapes arr1 to new size (equal number of items)

Subject	**Matlab**	**Python/Numpy-Arrays/DataObjects**
Data Copying	Matlab always uses deep copying. *b = a* -> *b* and *a* contain separated data in memory	Python usually creates shallow copies (deep copy only if necessary). Therefore *a* and *b* share the same data.
Indexing	Matlab uses one-based indexing	Python always uses zero-based indexing
Ranges	1:4 means the items at one-based indices [1,2,3,4] Both boundaries are included in the range.	In Python the same is achieved by 0:4 -> [0,1,2,3] The second boundary is always excluded!