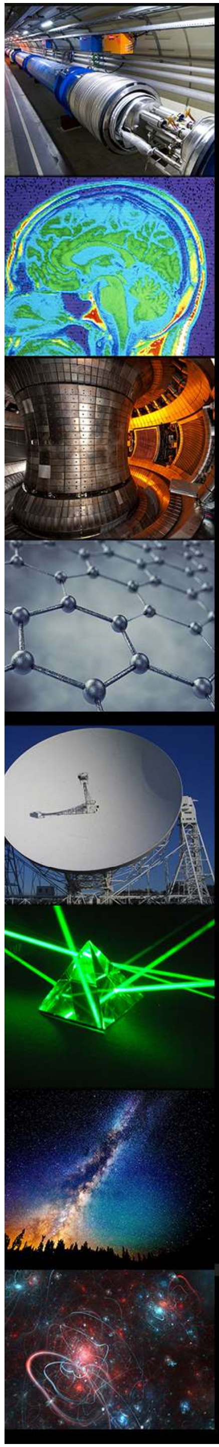
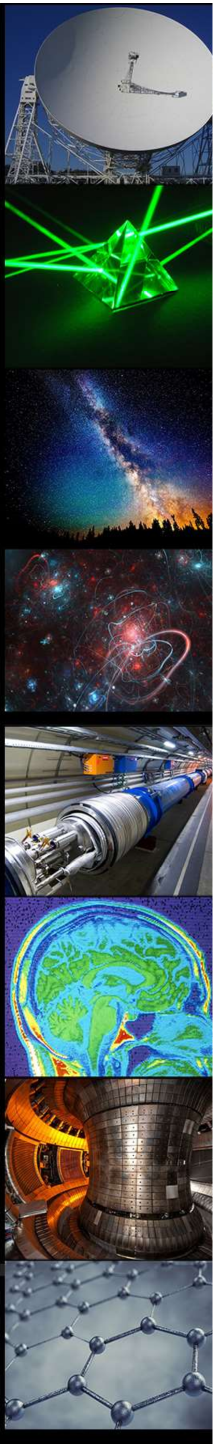
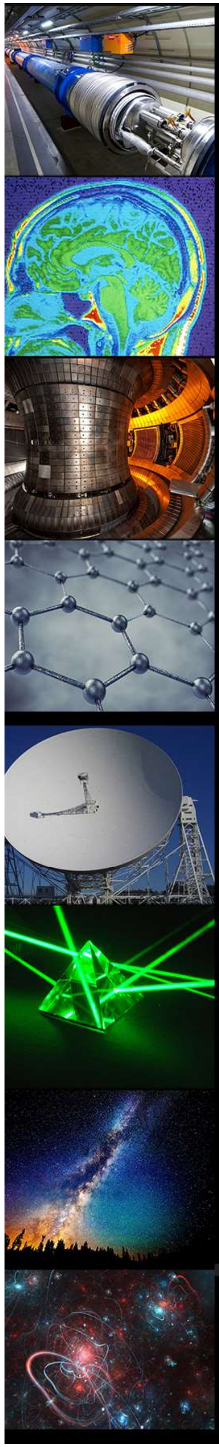


Lecture 2 Introduction to Python

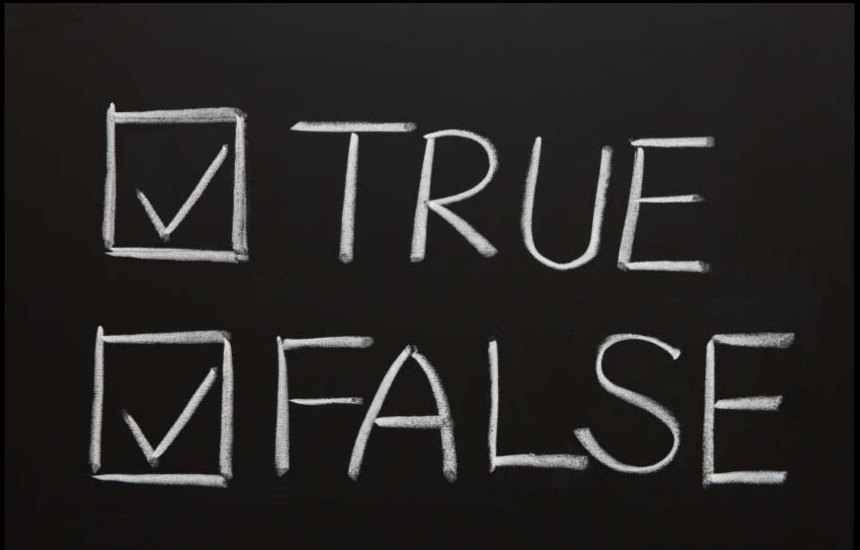
lecturer Alexander Gorbunov
Email: agorbunov@hse.ru

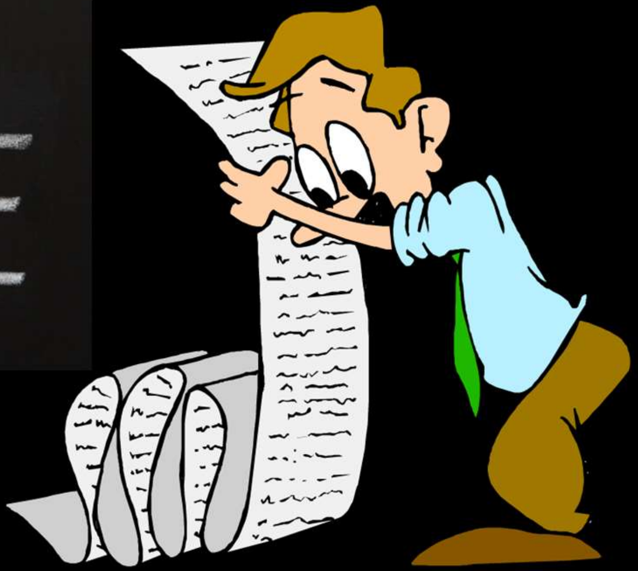


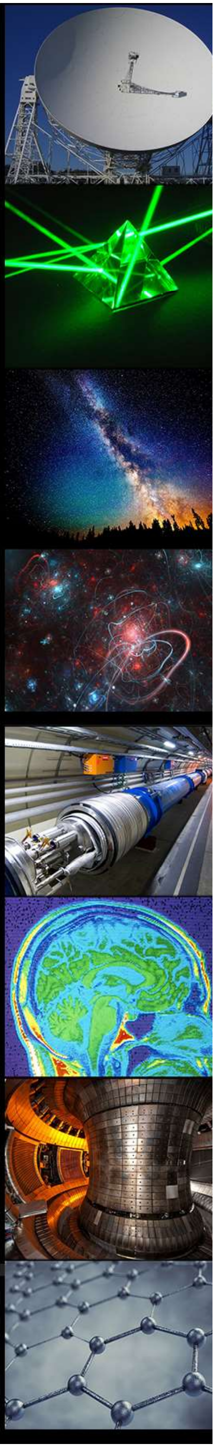


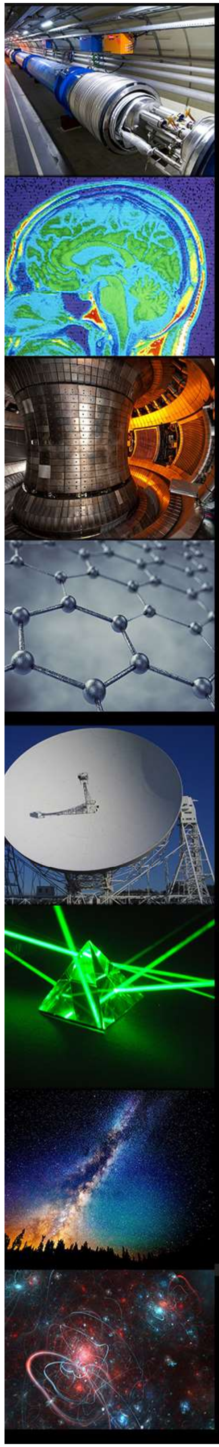
1.4 Basic Objects II:

Booleans, Tuples, and Dictionaries

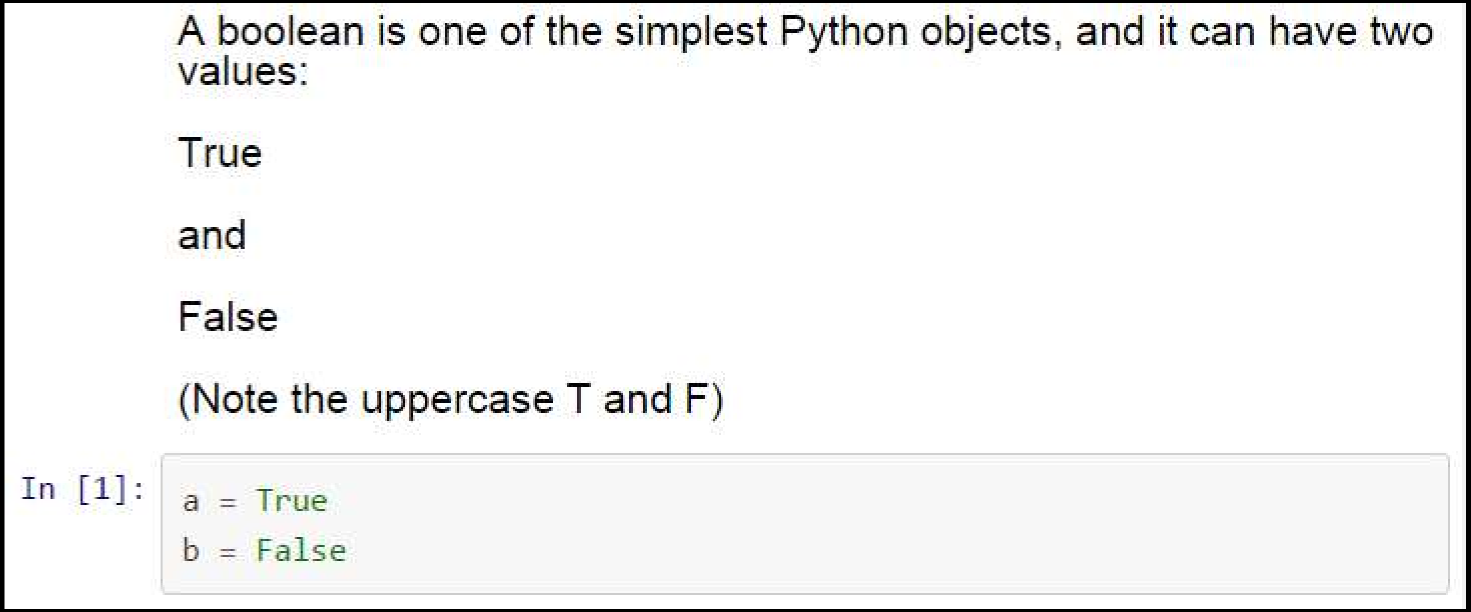








Booleans 1



A boolean is one of the simplest Python objects, and it can have two values:

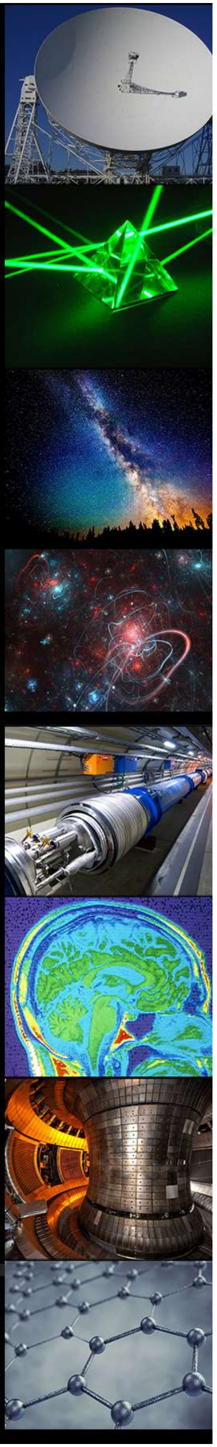
True

and

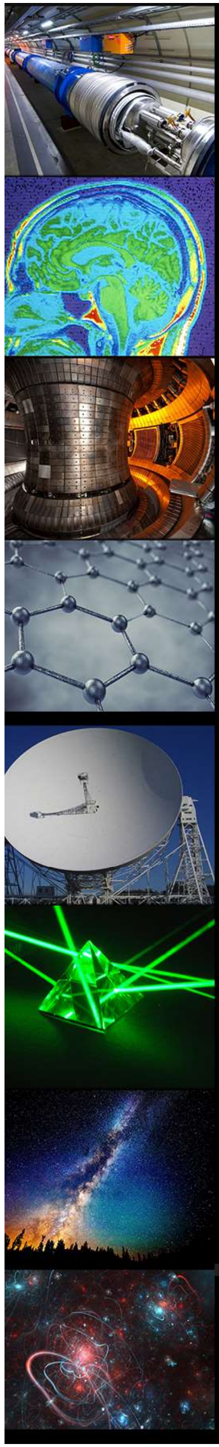
False

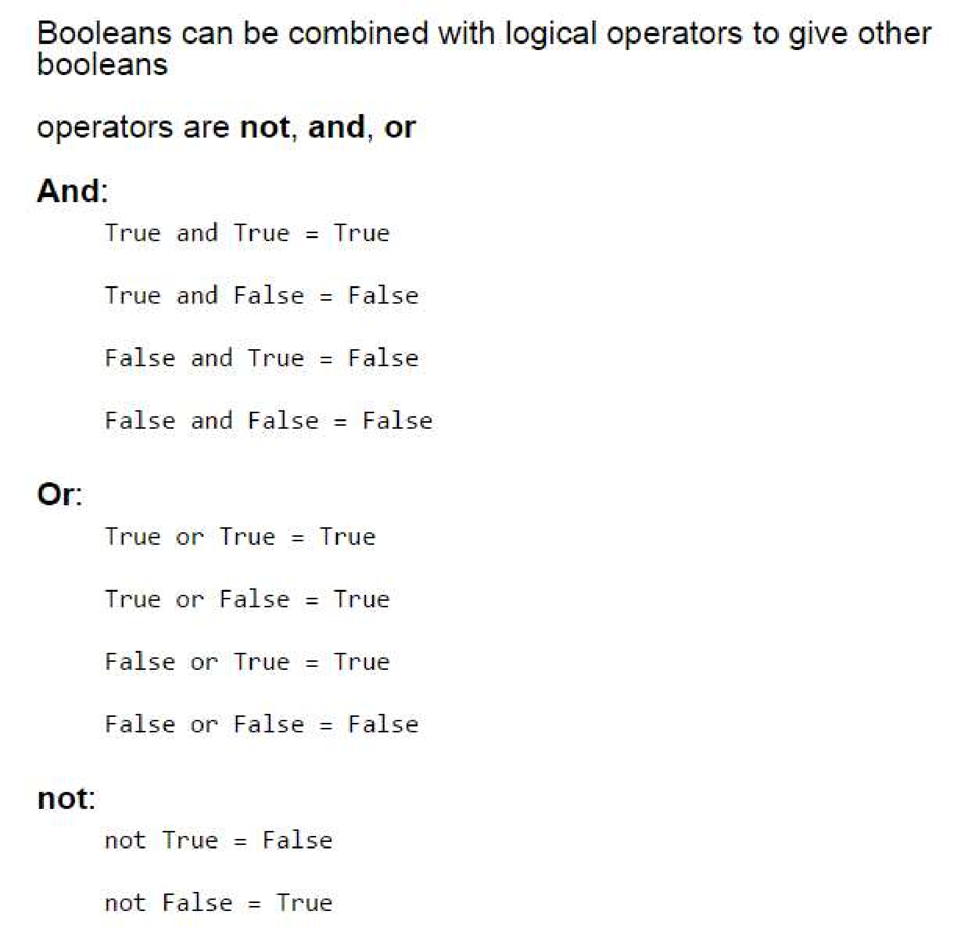
(Note the uppercase T and F)

In [1]:
```
a = True
b = False
```



Booleans 2





Booleans can be combined with logical operators to give other booleans

operators are **not**, **and**, **or**

And:

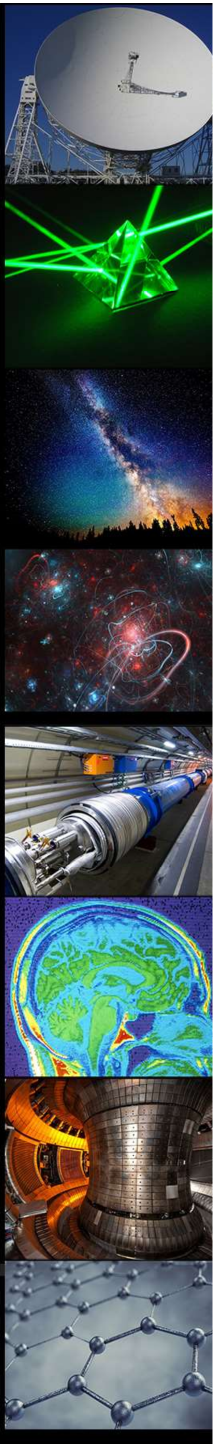
```
True and True = True

True and False = False

False and True = False

False and False = False
```

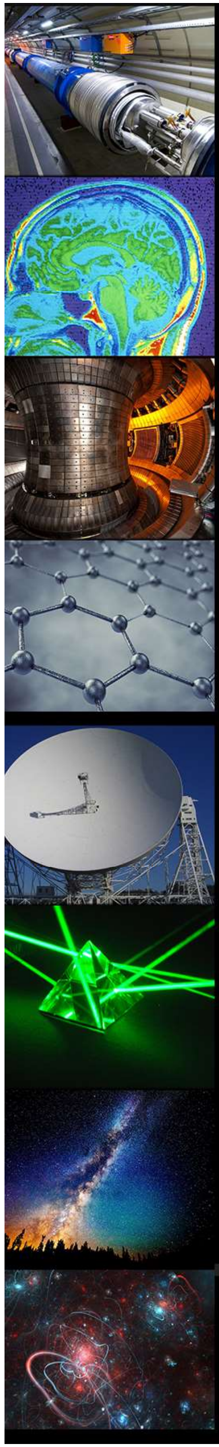
Or:

```
True or True = True

True or False = True

False or True = True

False or False = False
```

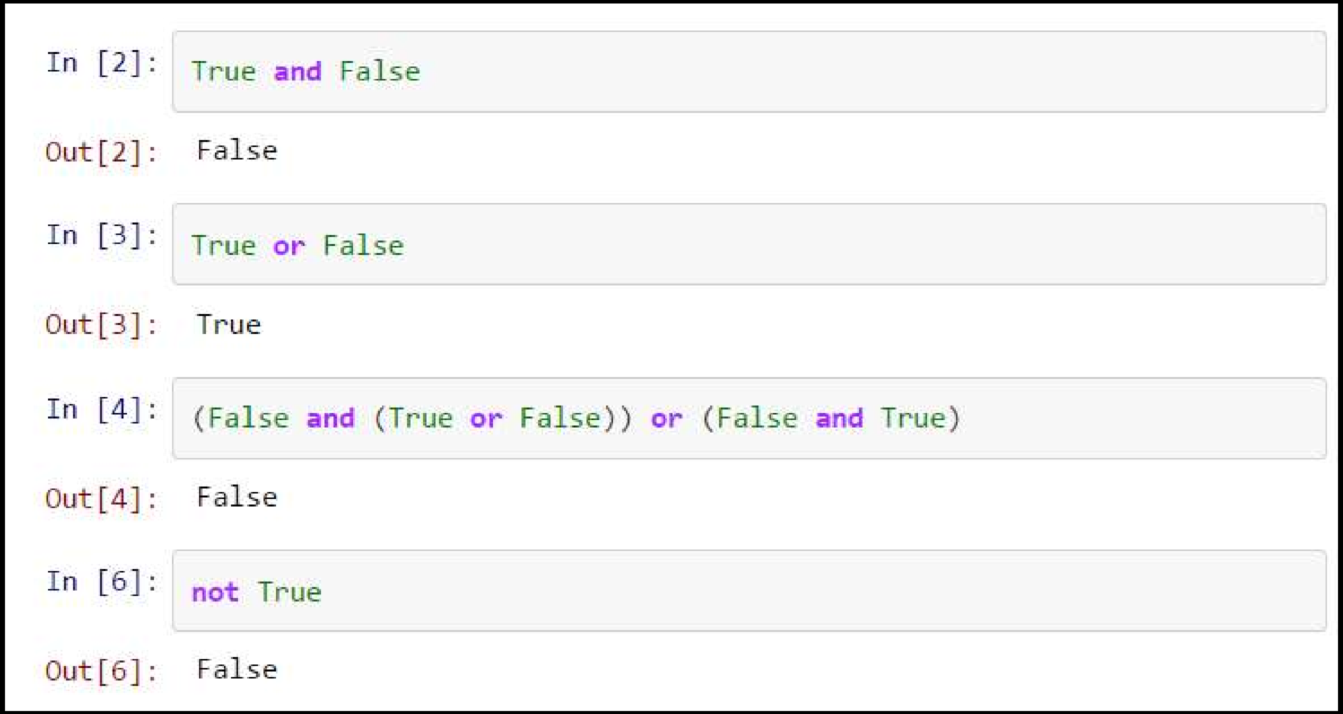
not:

```
not True = False

not False = True
```

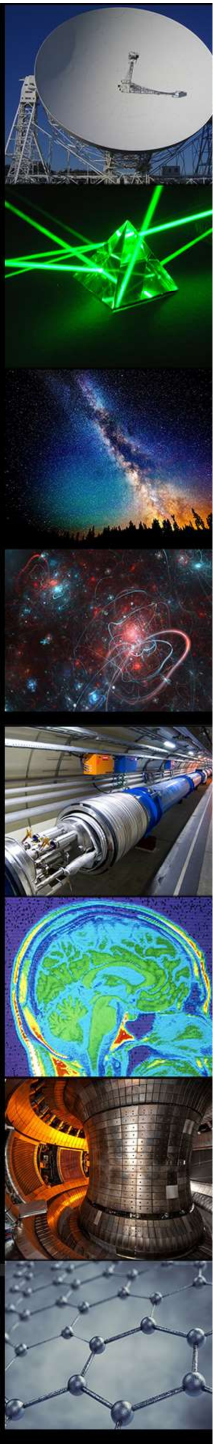


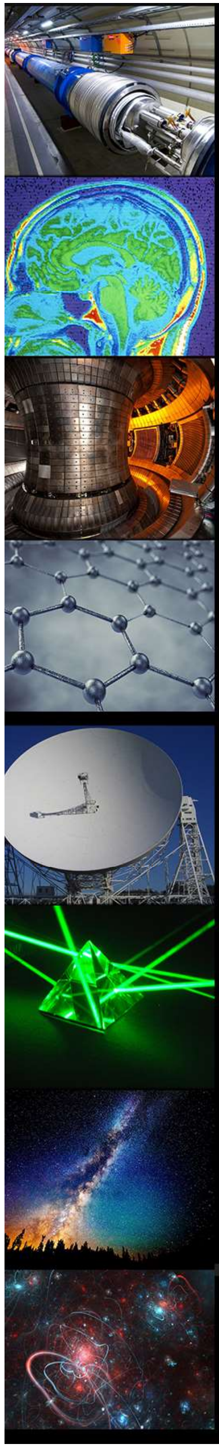


Booleans 3

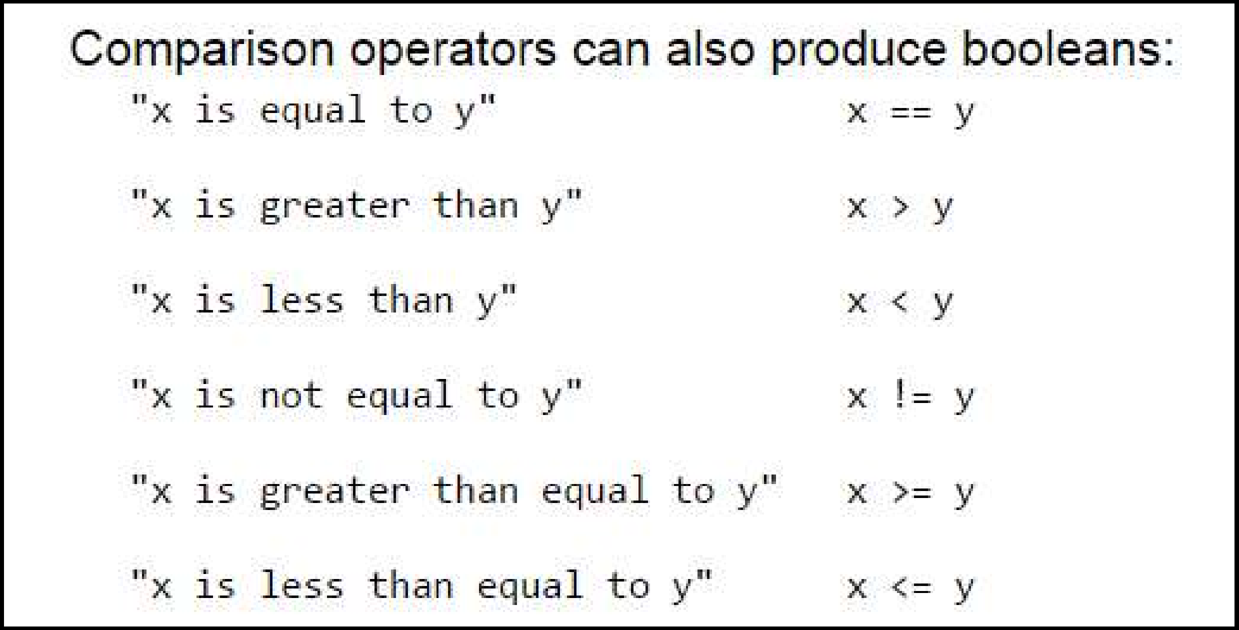


```
In [2]:  True and False

Out[2]:  False

In [3]:  True or False

Out[3]:  True

In [4]:  (False and (True or False)) or (False and True)

Out[4]:  False

In [6]:  not True

Out[6]:  False
```



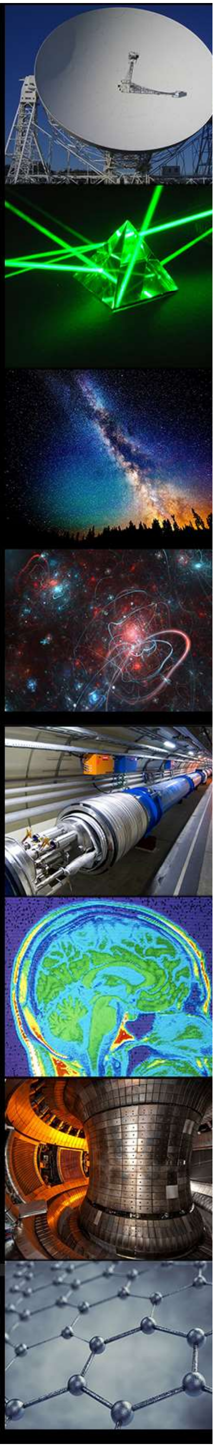


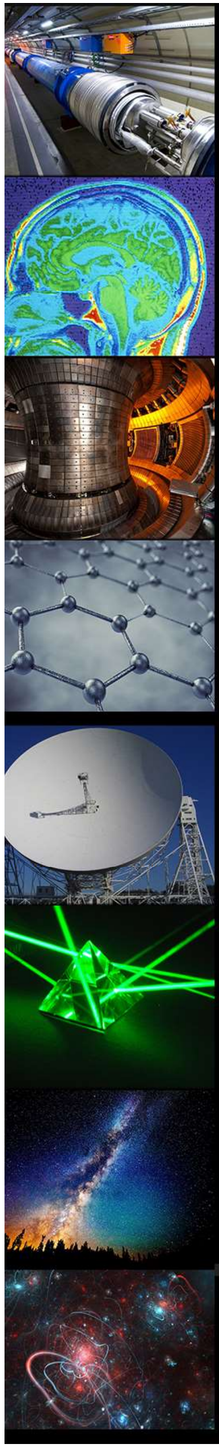
Booleans 4



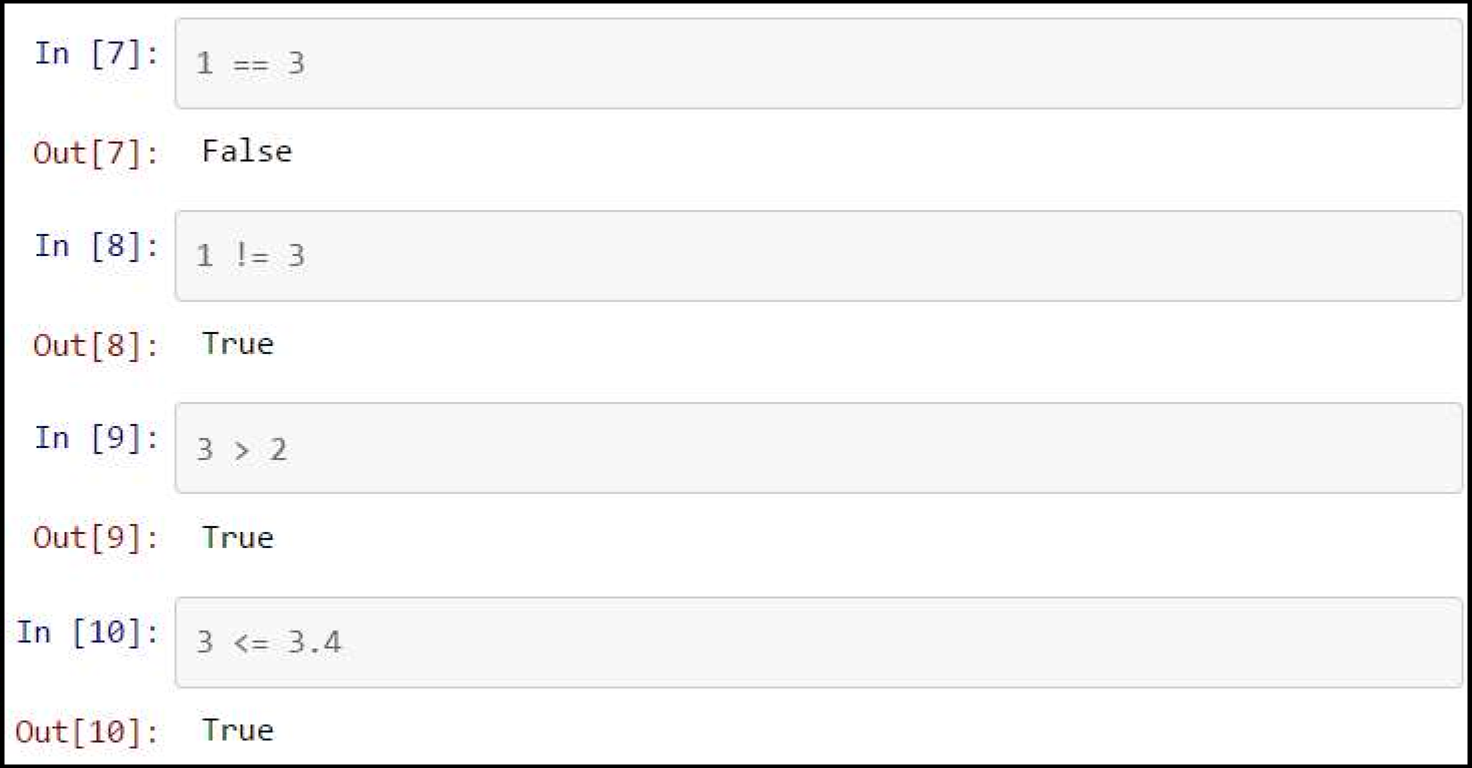
Comparison operators can also produce booleans:

"x is equal to y"	x == y
"x is greater than y"	x > y
"x is less than y"	x < y
"x is not equal to y"	x != y
"x is greater than equal to y"	x >= y
"x is less than equal to y"	x <= y

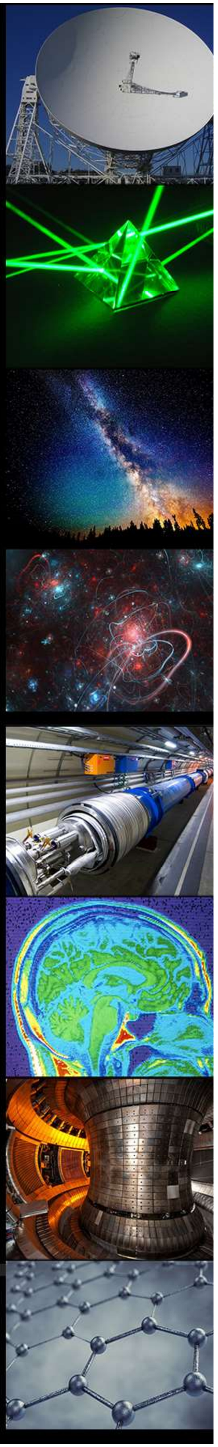




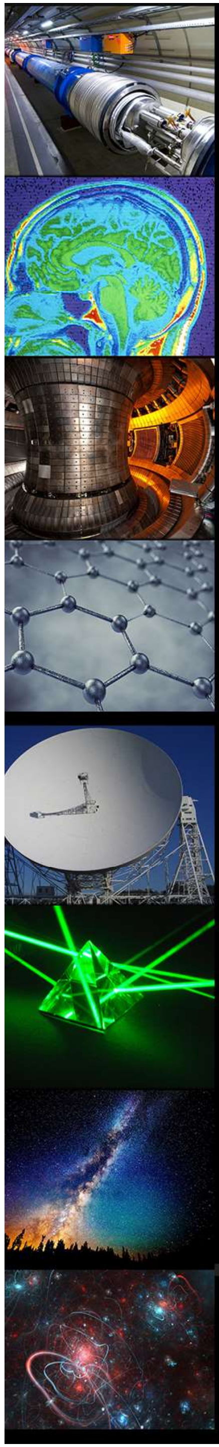
Booleans 5

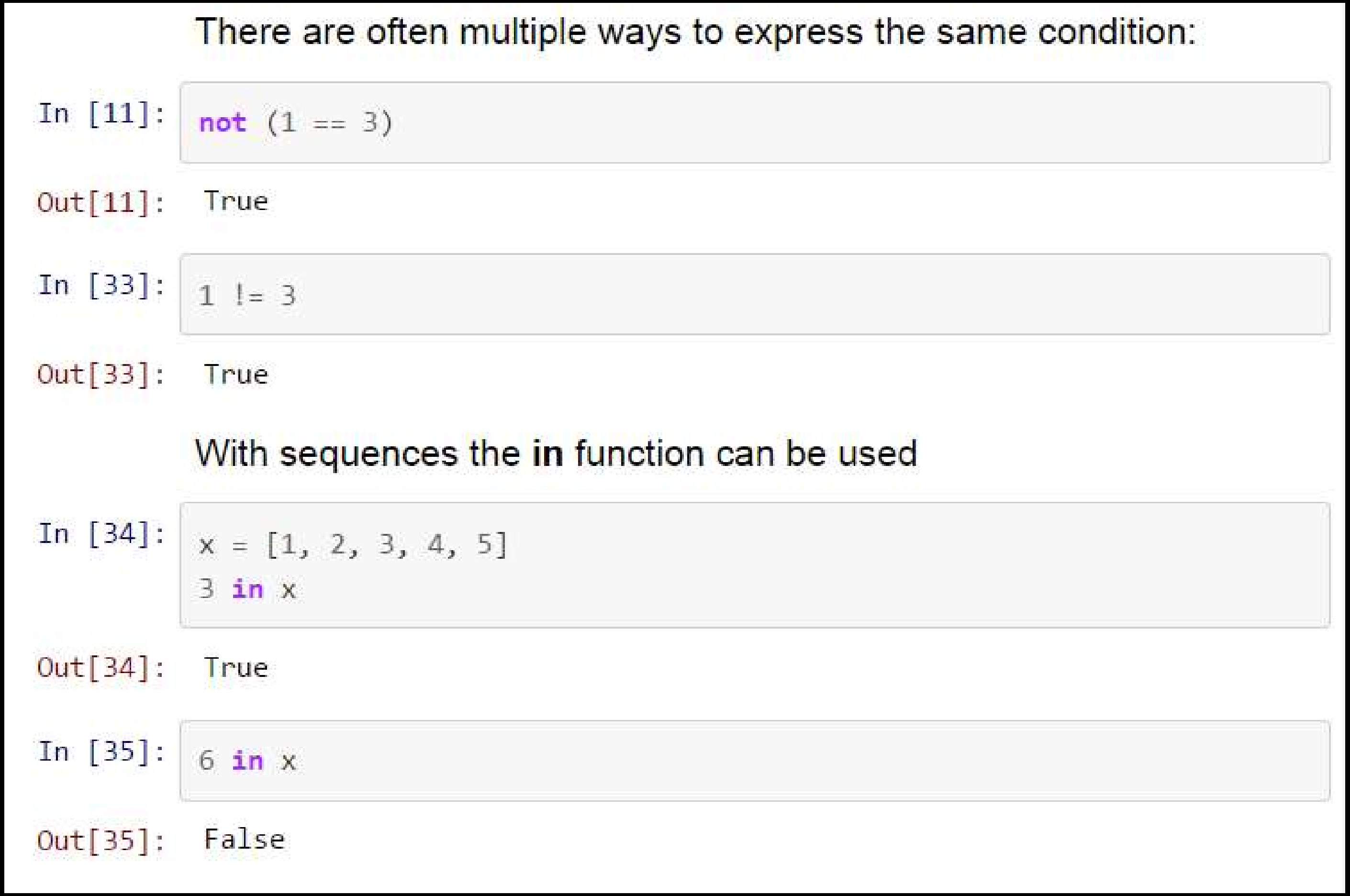


```
In [7]:  1 == 3

Out[7]:  False

In [8]:  1 != 3

Out[8]:  True

In [9]:  3 > 2

Out[9]:  True

In [10]: 3 <= 3.4

Out[10]: True
```



Booleans 6



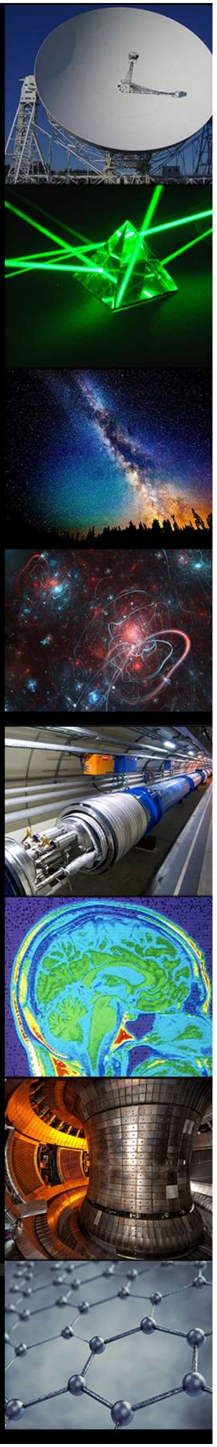


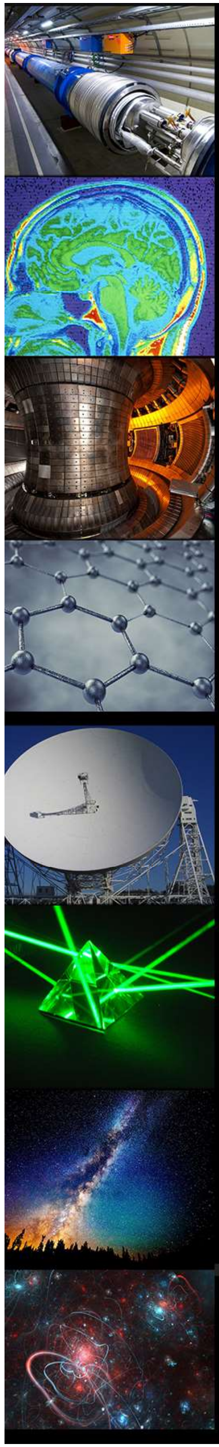
There are often multiple ways to express the same condition:

```
In [11]: not (1 == 3)

Out[11]: True

In [33]: 1 != 3

Out[33]: True
```

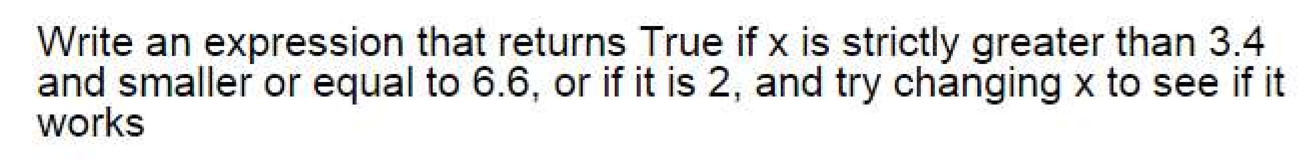
With sequences the **in** function can be used

```
In [34]: x = [1, 2, 3, 4, 5]
         3 in x

Out[34]: True

In [35]: 6 in x

Out[35]: False
```

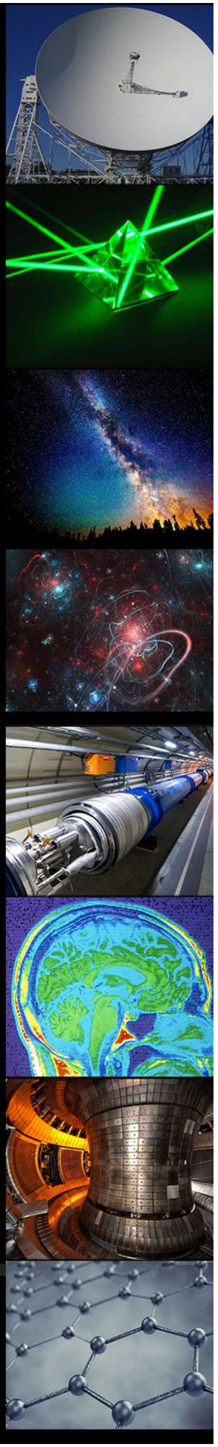


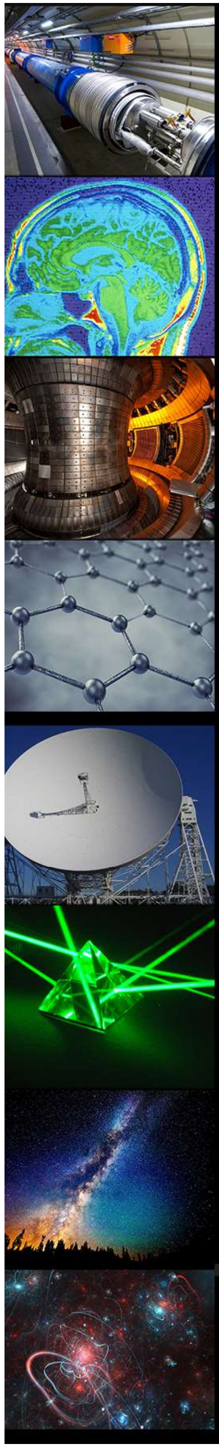


Exercise 1.4a

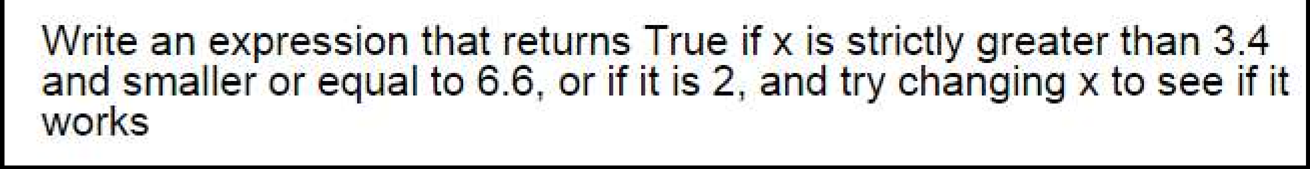


Write an expression that returns True if x is strictly greater than 3.4 and smaller or equal to 6.6, or if it is 2, and try changing x to see if it works

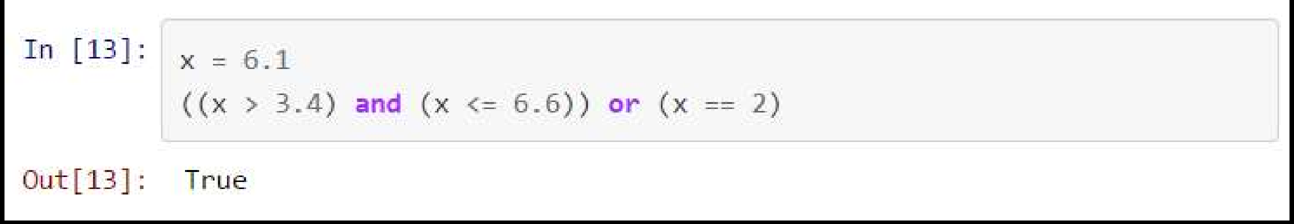




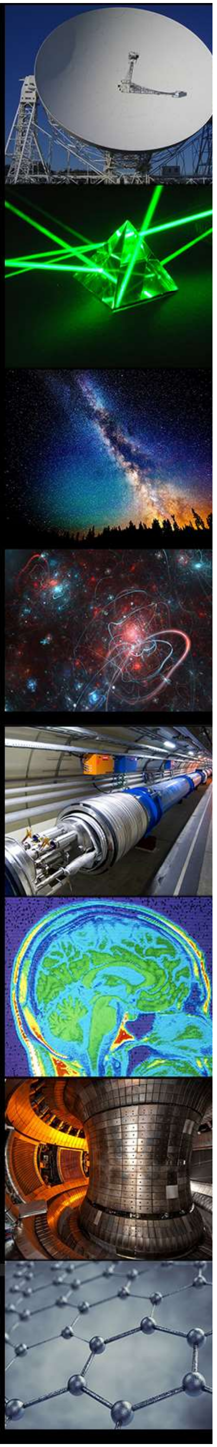
Solution 1.4a



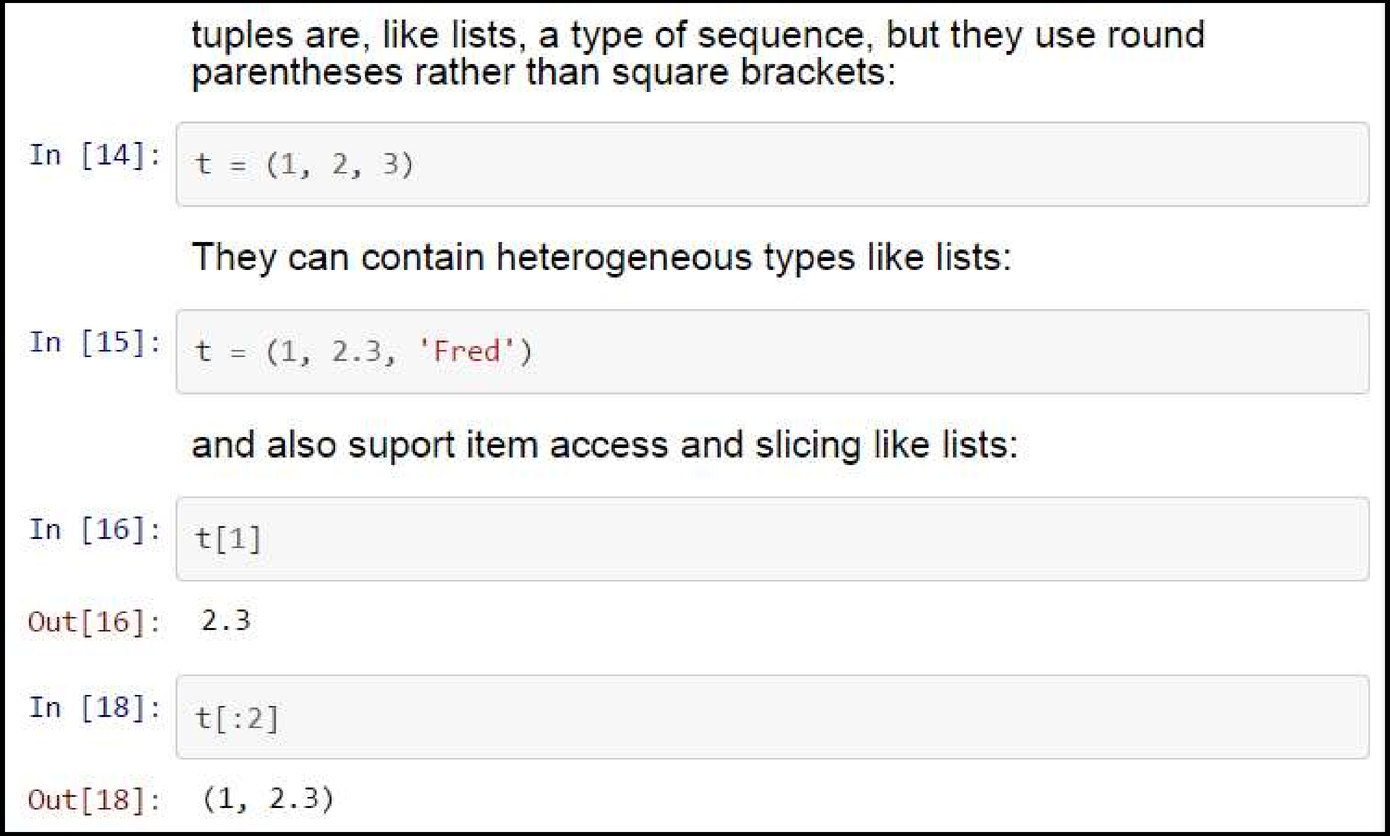
Write an expression that returns True if x is strictly greater than 3.4 and smaller or equal to 6.6, or if it is 2, and try changing x to see if it works



```
In [13]:  x = 6.1
          ((x > 3.4) and (x <= 6.6)) or (x == 2)

Out[13]:  True
```



Tuples 1



tuples are, like lists, a type of sequence, but they use round parentheses rather than square brackets:

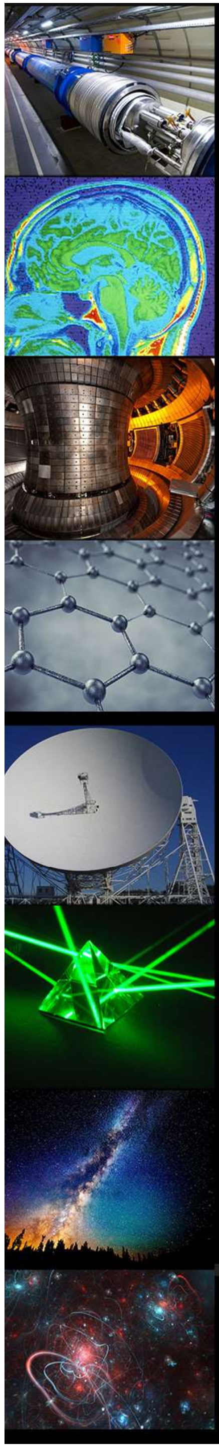
```
In [14]: t = (1, 2, 3)
```

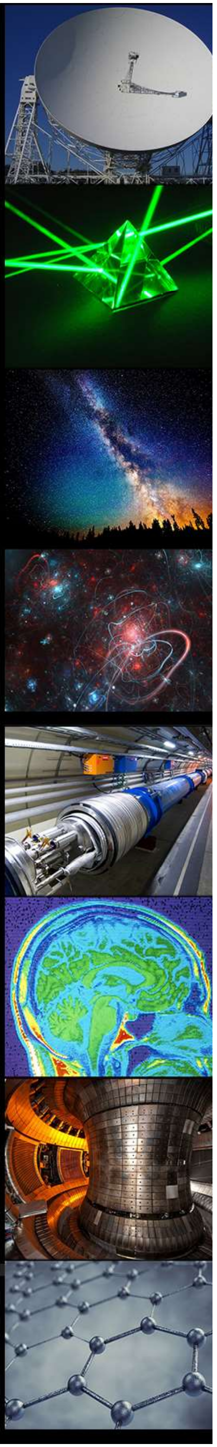
They can contain heterogeneous types like lists:

```
In [15]: t = (1, 2.3, 'Fred')
```

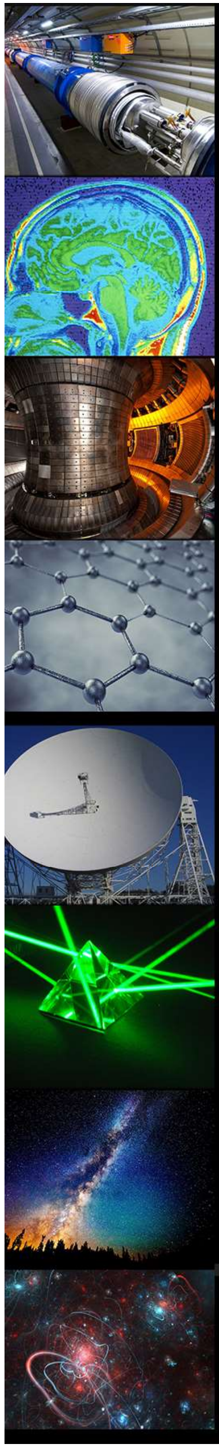
and also suport item access and slicing like lists:

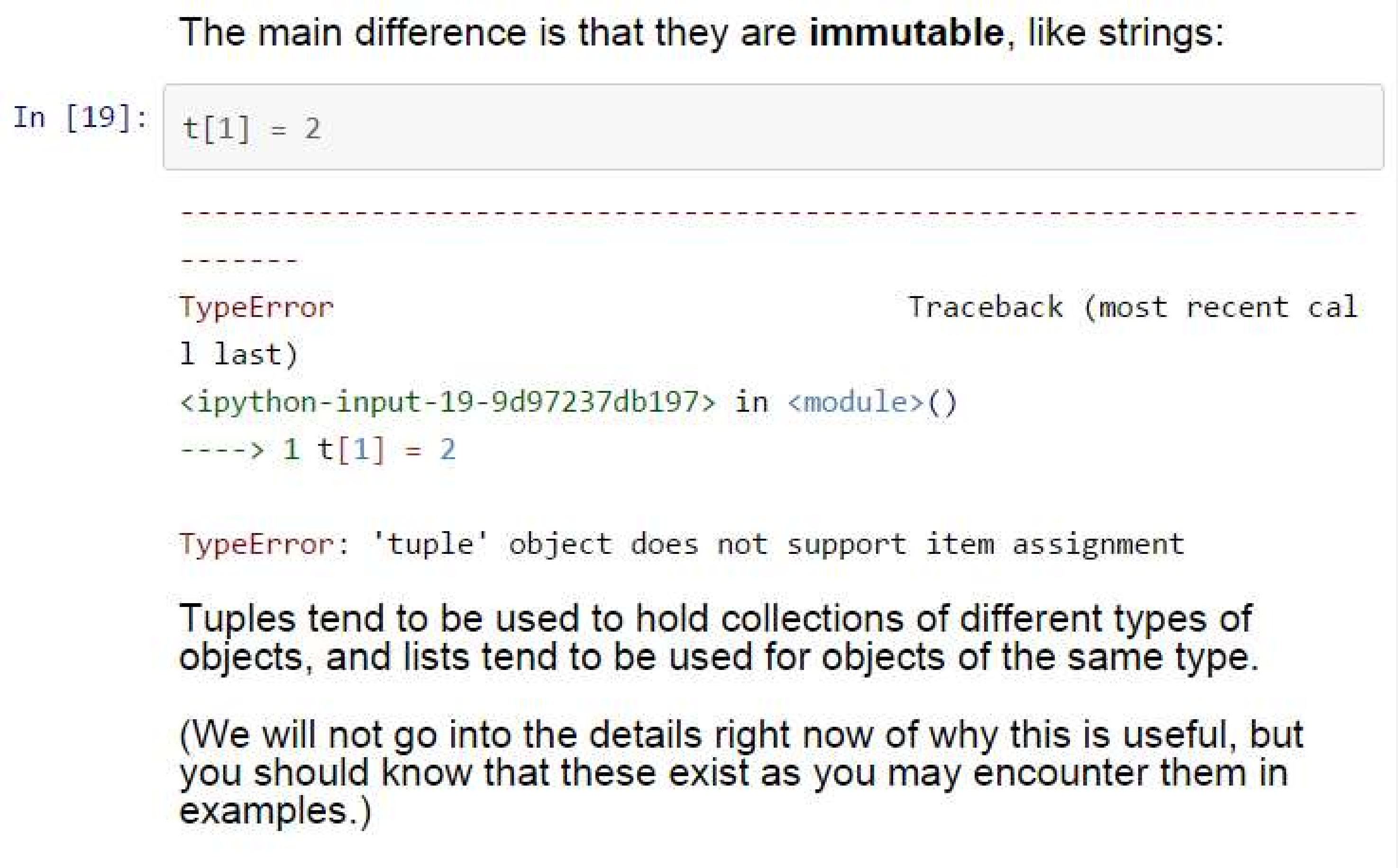
```
In [16]: t[1]

Out[16]: 2.3

In [18]: t[:2]

Out[18]: (1, 2.3)
```





Tuples 2





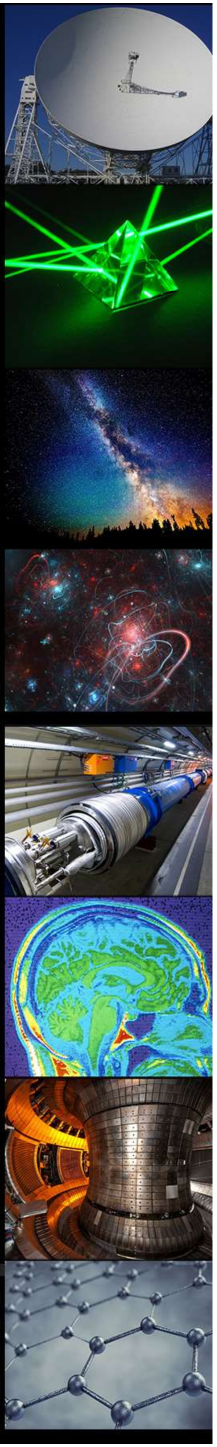
The main difference is that they are **immutable**, like strings:

```
In [19]: t[1] = 2
```

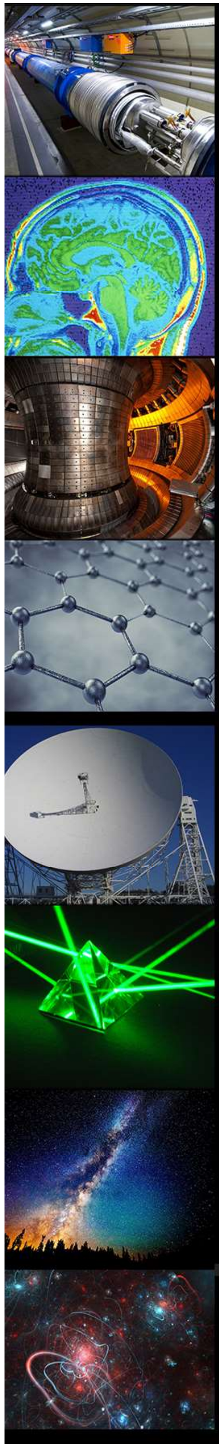
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-9d97237db197> in <module>()
----> 1 t[1] = 2

TypeError: 'tuple' object does not support item assignment
```

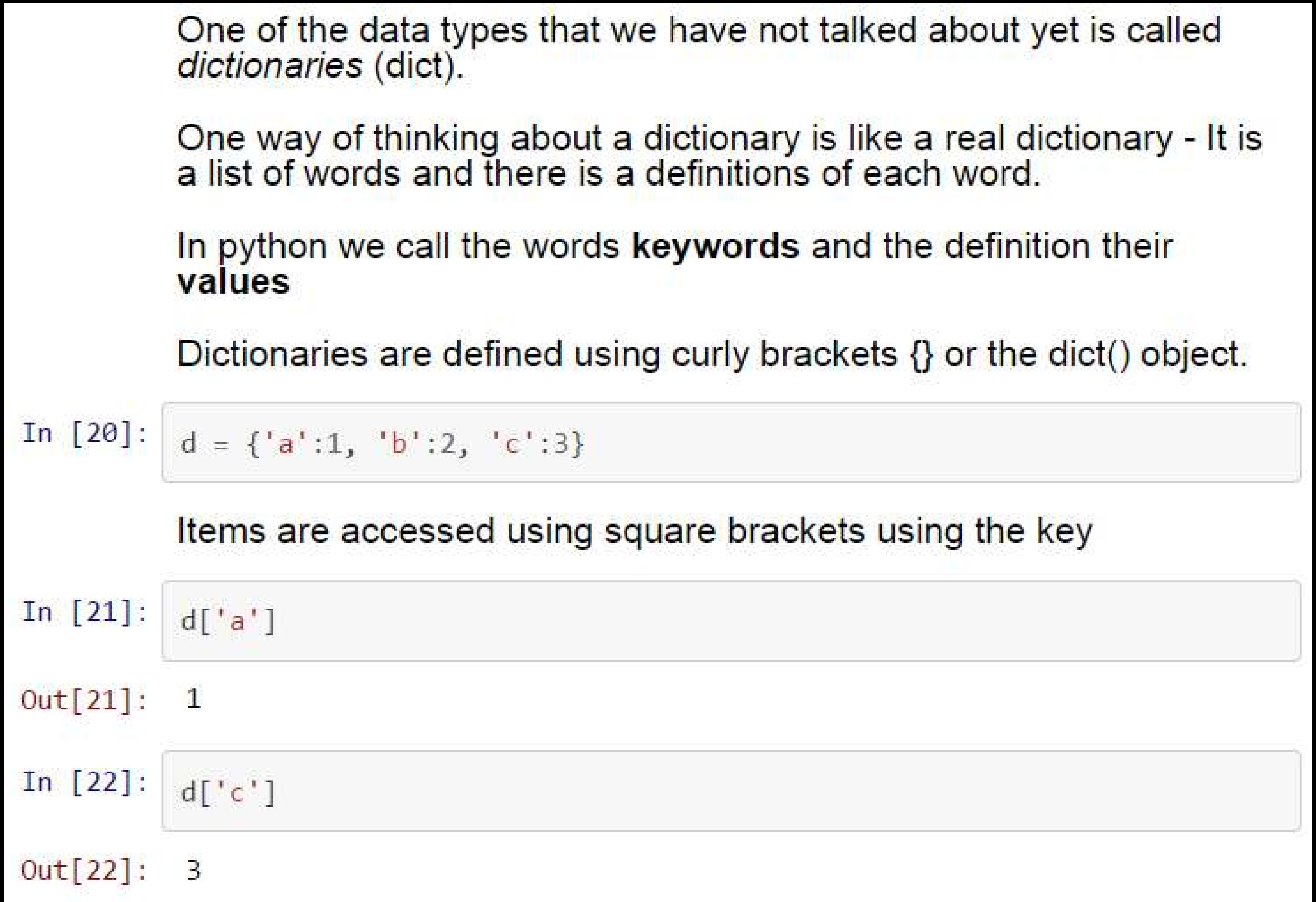
Tuples tend to be used to hold collections of different types of objects, and lists tend to be used for objects of the same type.

(We will not go into the details right now of why this is useful, but you should know that these exist as you may encounter them in examples.)



Dictionaries 1





One of the data types that we have not talked about yet is called *dictionaries* (dict).

One way of thinking about a dictionary is like a real dictionary - It is a list of words and there is a definitions of each word.

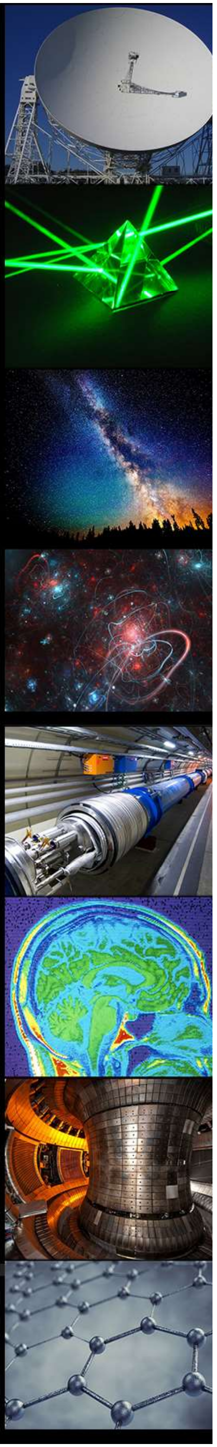
In python we call the words **keywords** and the definition their **values**

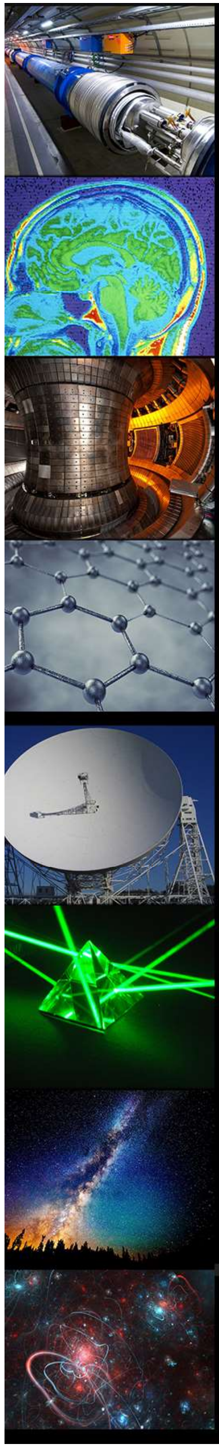
Dictionaries are defined using curly brackets {} or the dict() object.

```
In [20]: d = {'a':1, 'b':2, 'c':3}
```

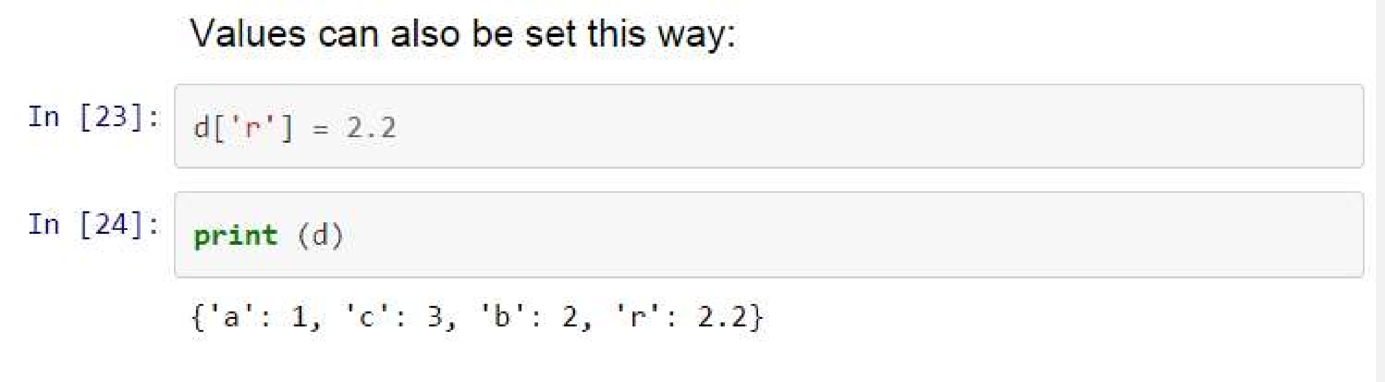
Items are accessed using square brackets using the key

```
In [21]: d['a']

Out[21]: 1

In [22]: d['c']

Out[22]: 3
```





Dictionaries 2

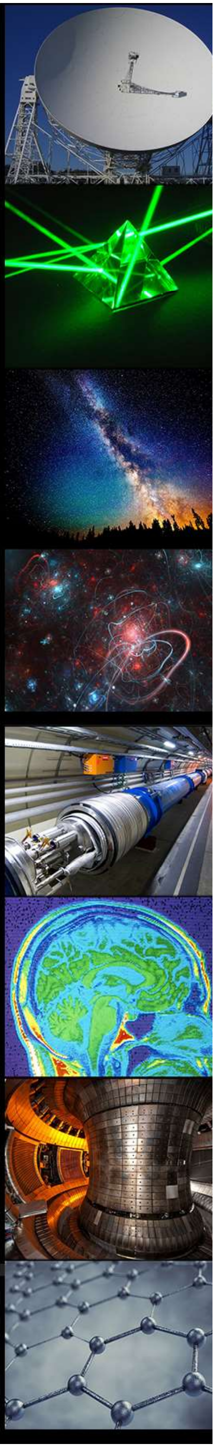


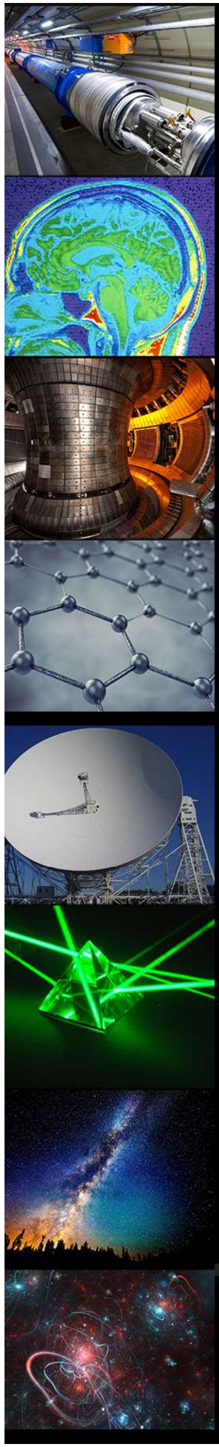
Values can also be set this way:

```
In [23]: d['r'] = 2.2
```

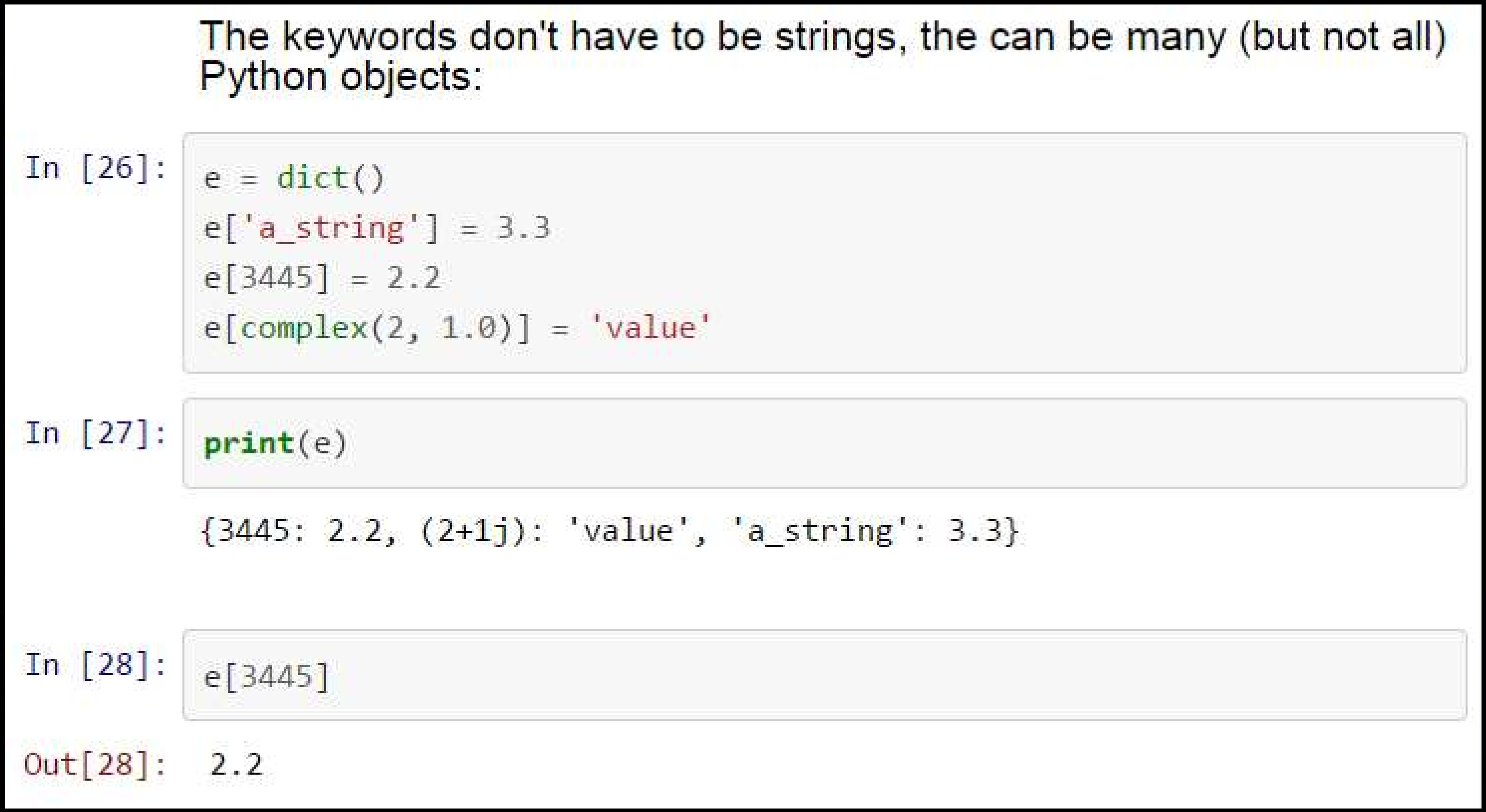
```
In [24]: print (d)
```

```
{'a': 1, 'c': 3, 'b': 2, 'r': 2.2}
```





Dictionaries 3



The keywords don't have to be strings, the can be many (but not all) Python objects:

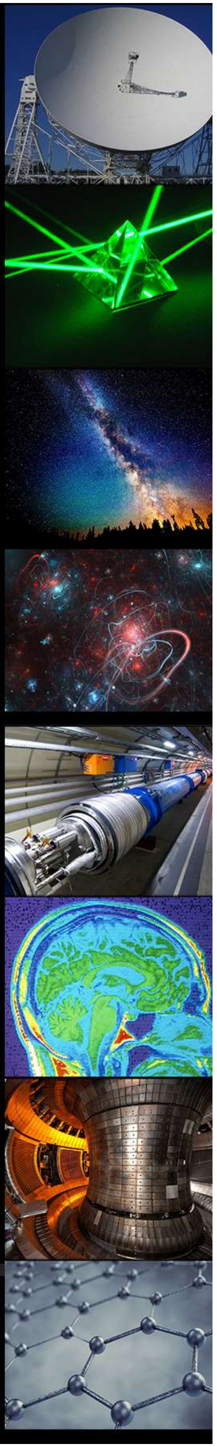
In [26]:
```
e = dict()
e['a_string'] = 3.3
e[3445] = 2.2
e[complex(2, 1.0)] = 'value'
```

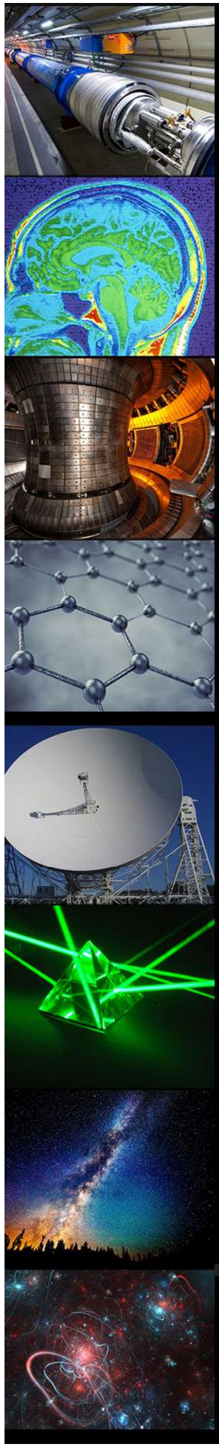
In [27]:
```
print(e)
```

```
{3445: 2.2, (2+1j): 'value', 'a_string': 3.3}
```

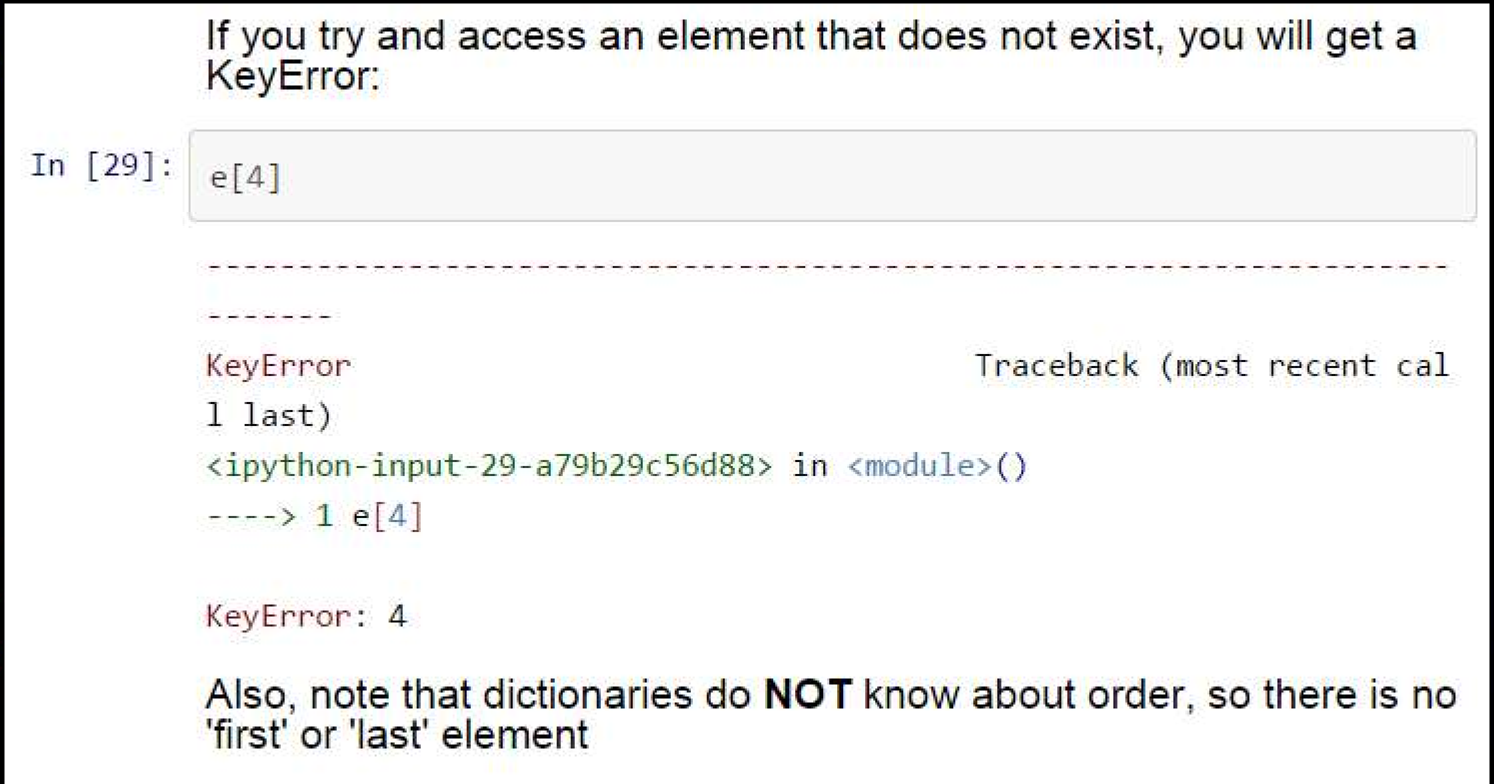
In [28]:
```
e[3445]
```

Out[28]:
```
2.2
```





Dictionaries 4

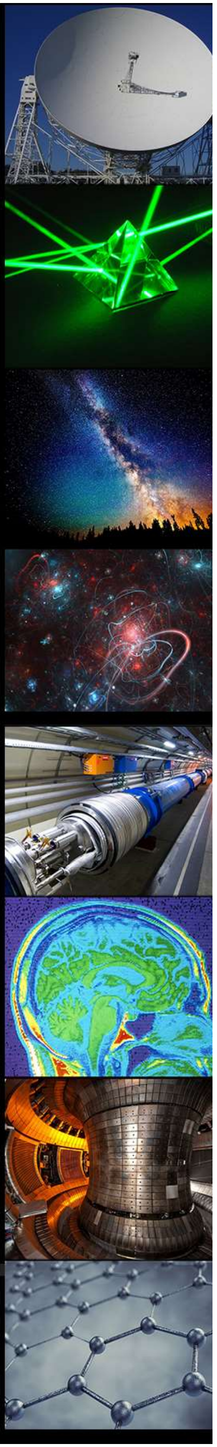


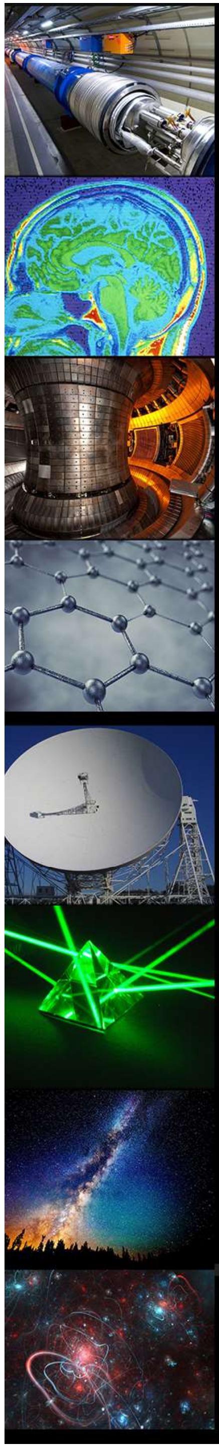
If you try and access an element that does not exist, you will get a KeyError:

```
In [29]: e[4]
```

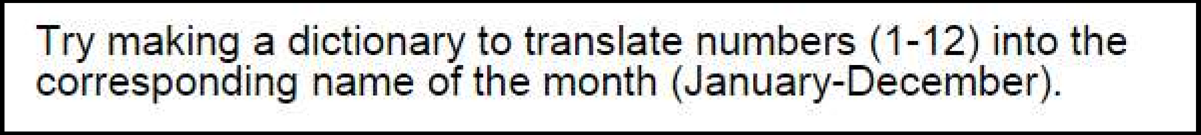
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent cal
l last)
<ipython-input-29-a79b29c56d88> in <module>()
----> 1 e[4]

KeyError: 4
```

Also, note that dictionaries do **NOT** know about order, so there is no 'first' or 'last' element

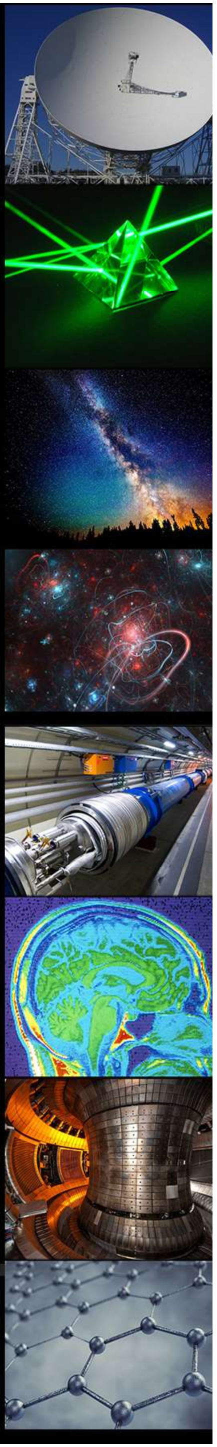


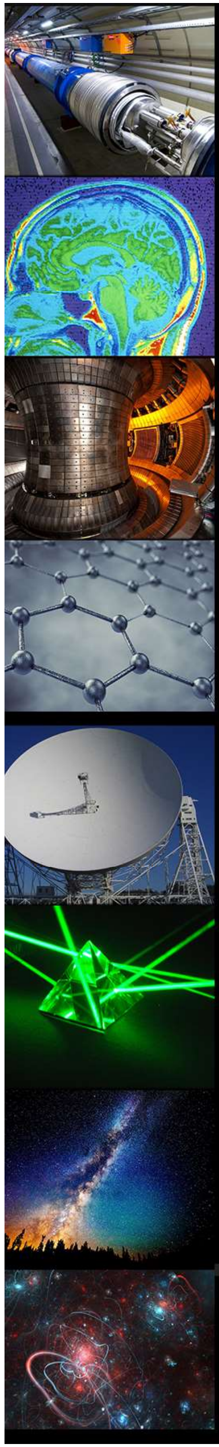


Exercise 1.4b



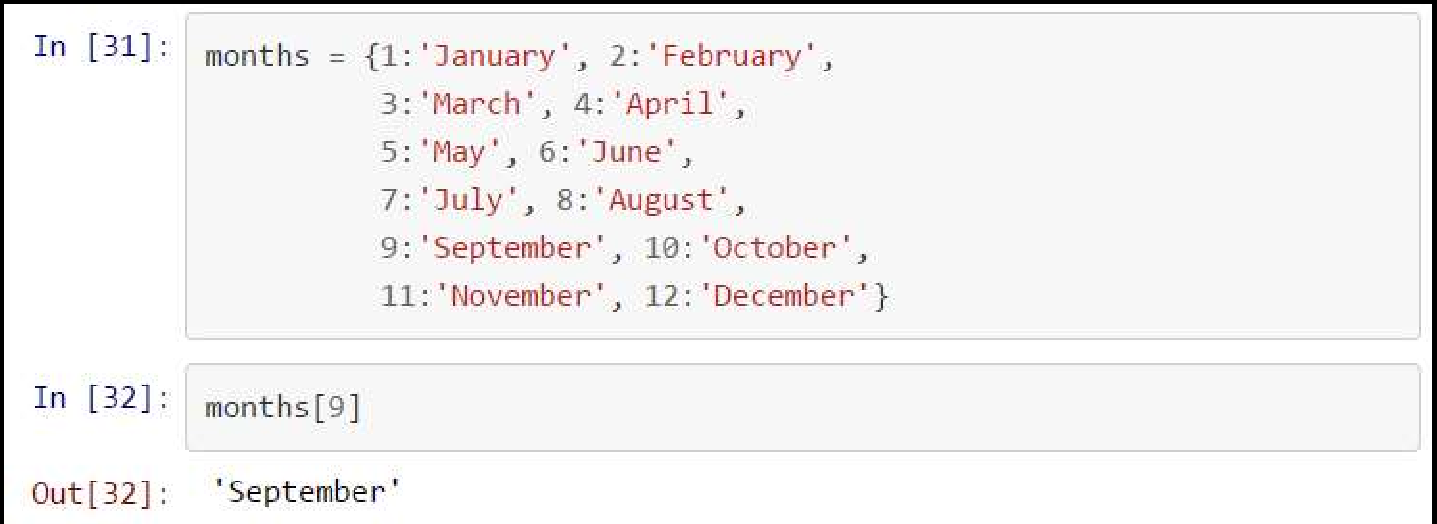
Try making a dictionary to translate numbers (1-12) into the corresponding name of the month (January-December).



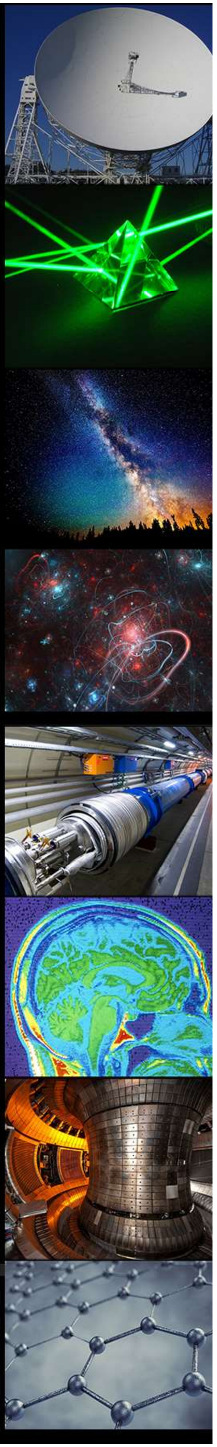


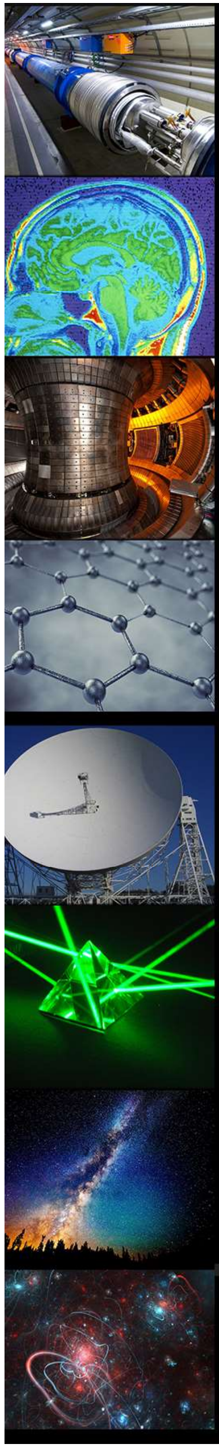
Solution 1.4b

Try making a dictionary to translate numbers (1-12) into the corresponding name of the month (January-December).

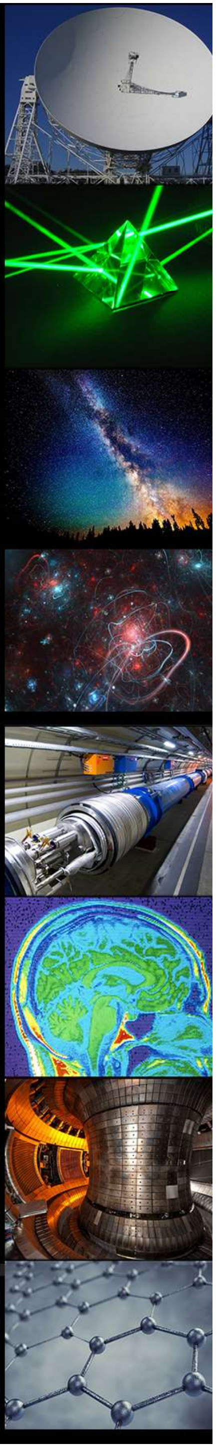


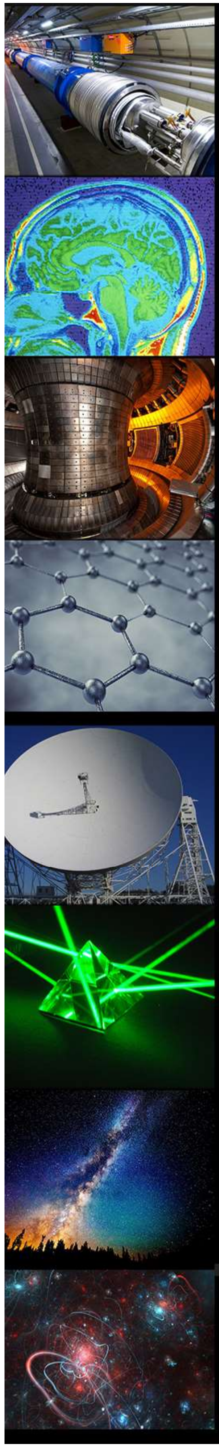
```
In [31]: months = {1:'January', 2:'February',
                   3:'March', 4:'April',
                   5:'May', 6:'June',
                   7:'July', 8:'August',
                   9:'September', 10:'October',
                   11:'November', 12:'December'}

In [32]: months[9]

Out[32]: 'September'
```



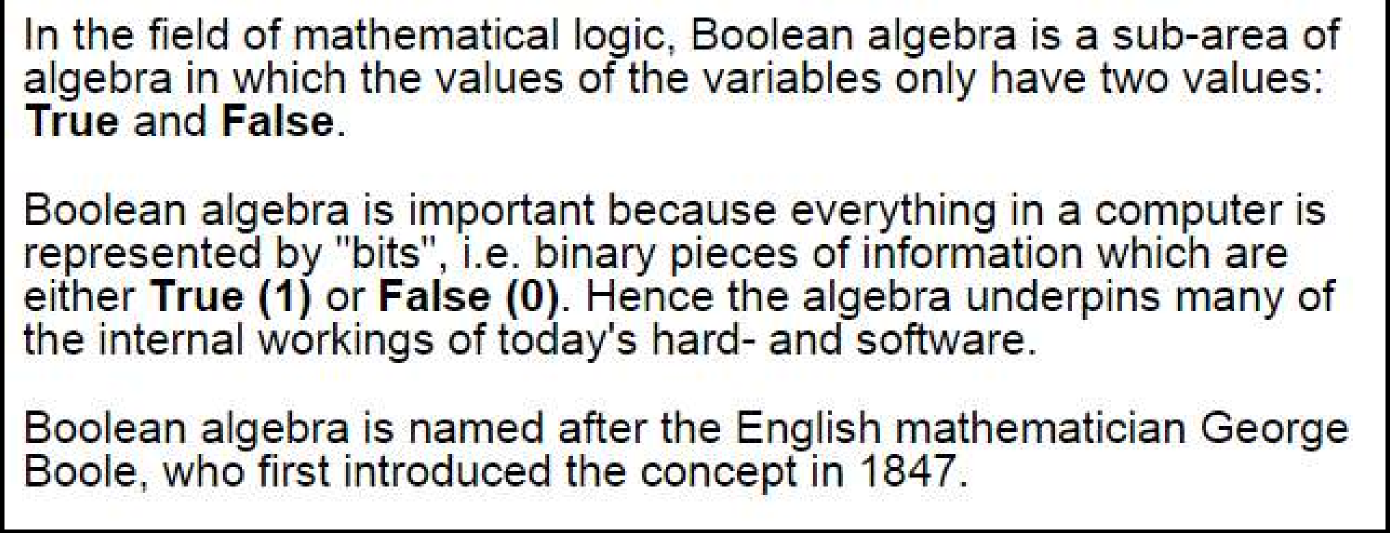


1.5 Boolean Algebra





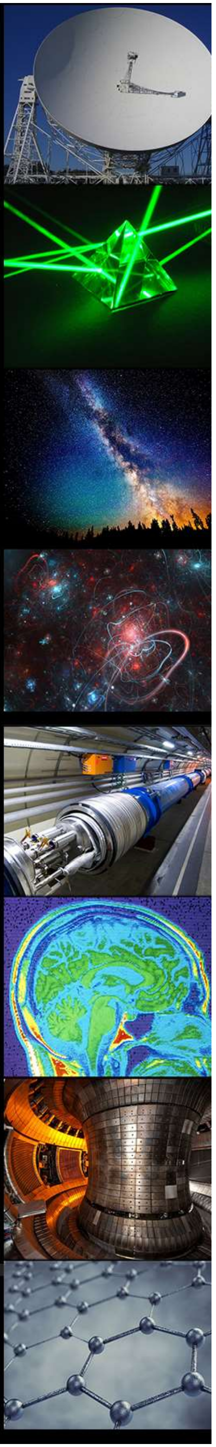
Boolean Algebra

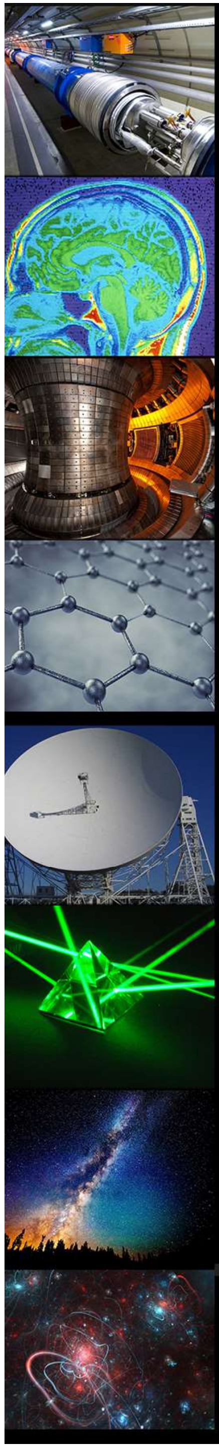


In the field of mathematical logic, Boolean algebra is a sub-area of algebra in which the values of the variables only have two values: **True** and **False**.

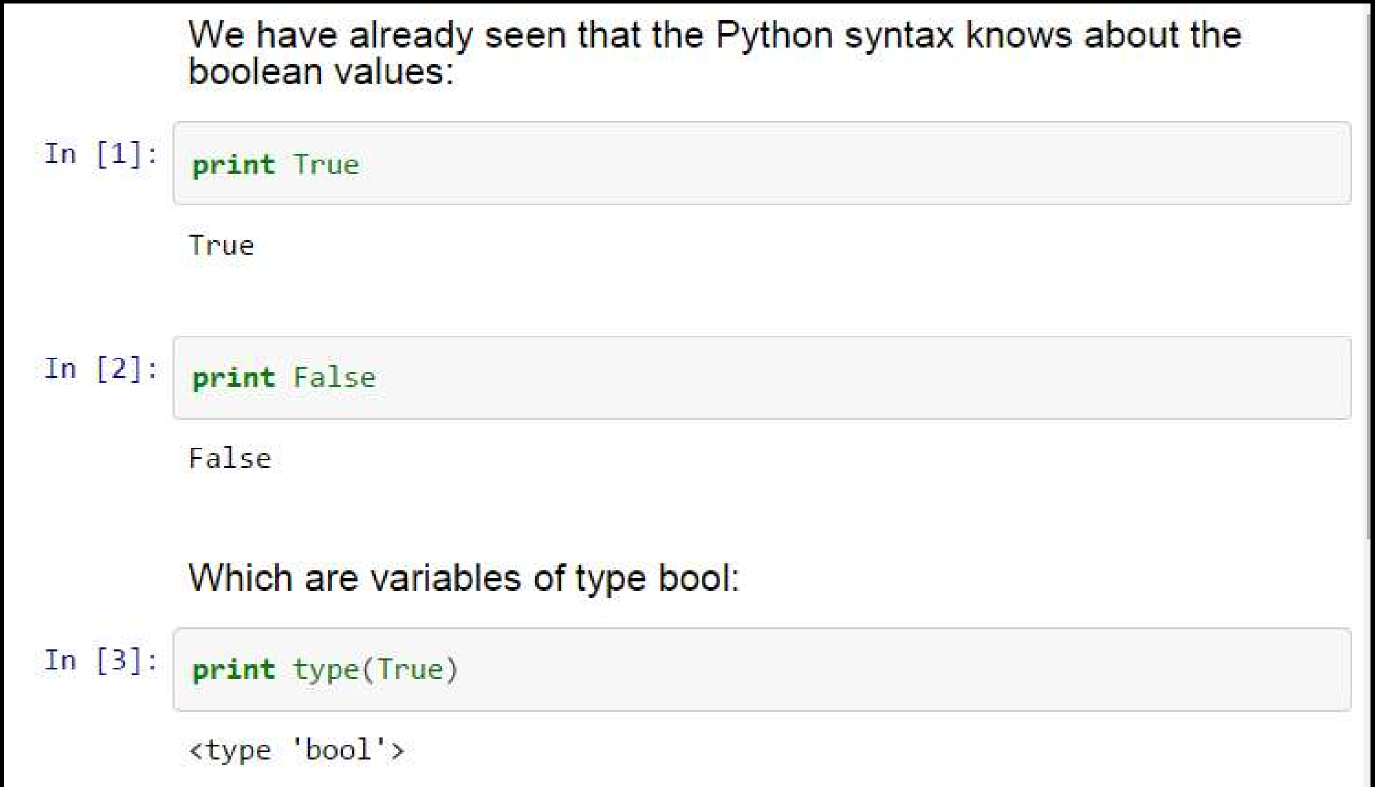
Boolean algebra is important because everything in a computer is represented by "bits", i.e. binary pieces of information which are either **True (1)** or **False (0)**. Hence the algebra underpins many of the internal workings of today's hard- and software.

Boolean algebra is named after the English mathematician George Boole, who first introduced the concept in 1847.





Booleans in Python 1



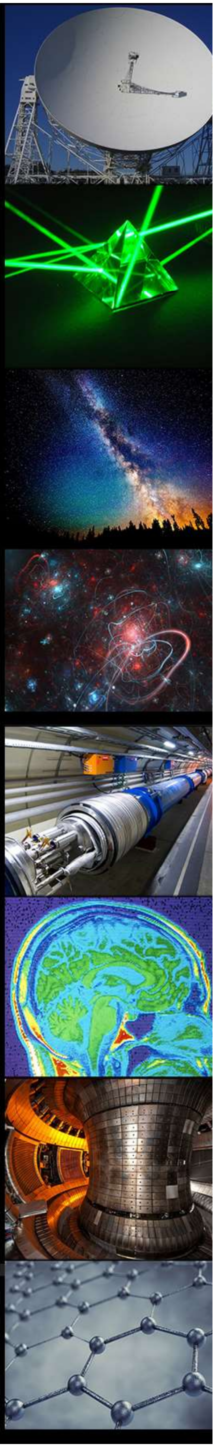
We have already seen that the Python syntax knows about the boolean values:

```
In [1]: print True

True
```

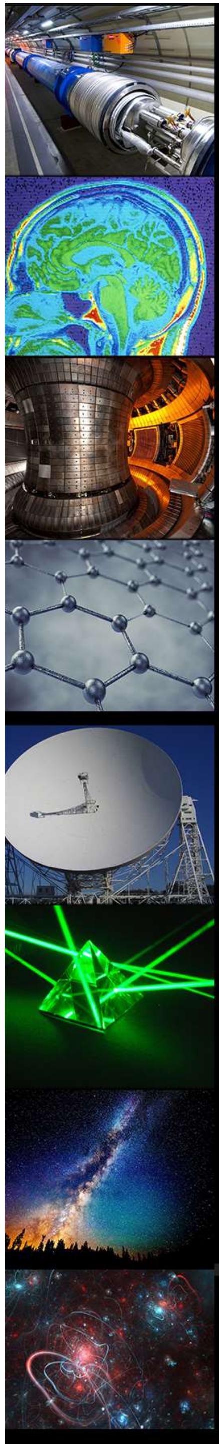
```
In [2]: print False

False
```

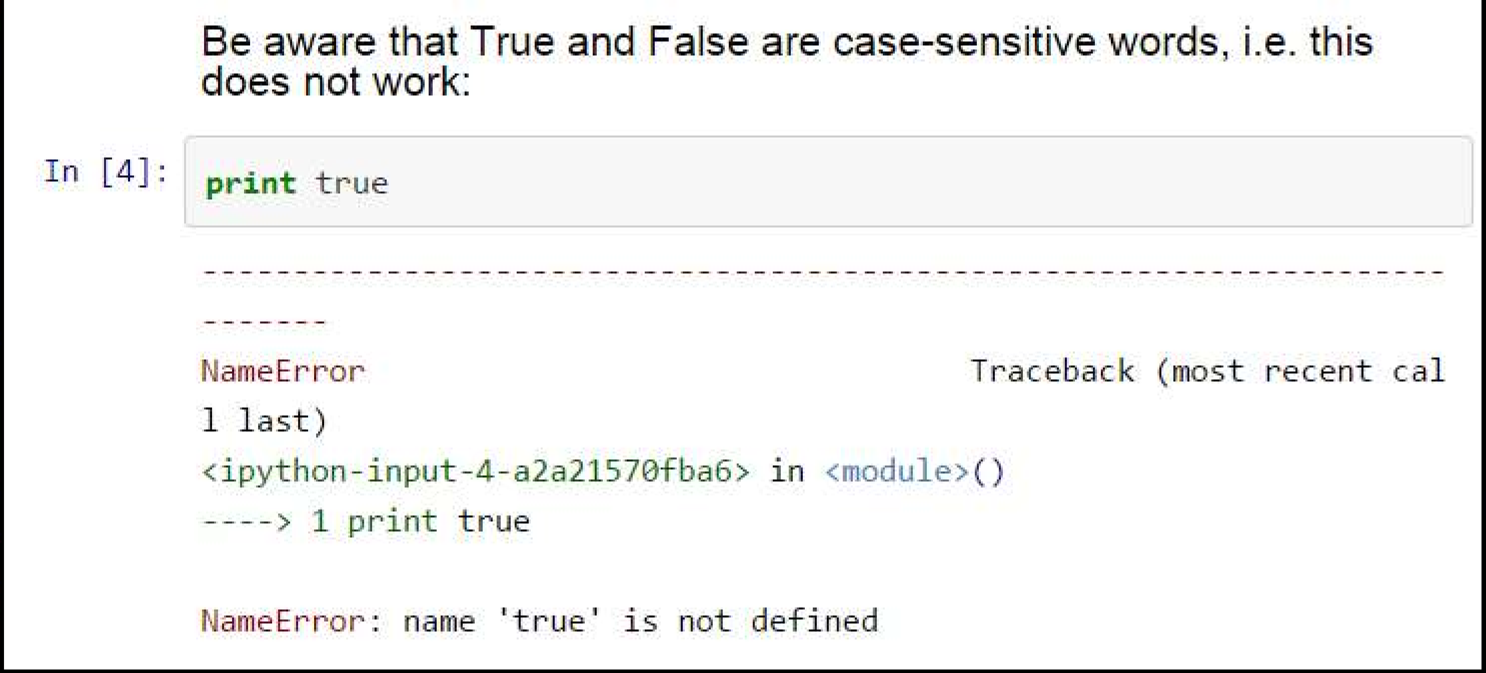
Which are variables of type bool:

```
In [3]: print type(True)

<type 'bool'>
```



Booleans in Python 2

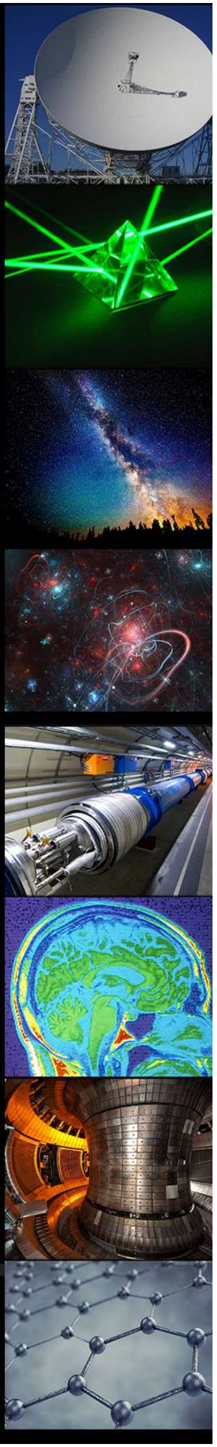


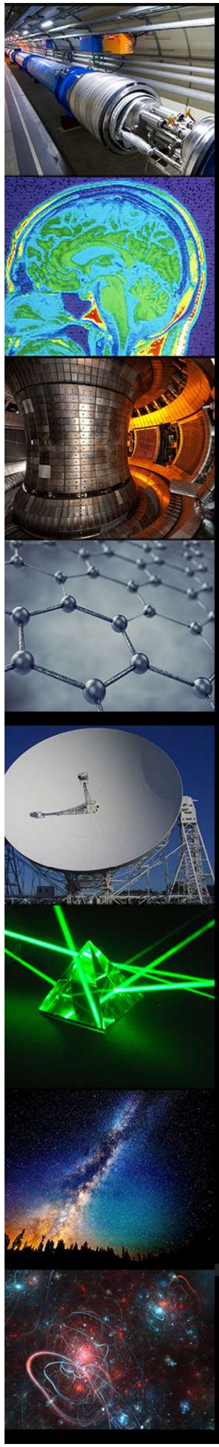


Be aware that True and False are case-sensitive words, i.e. this does not work:

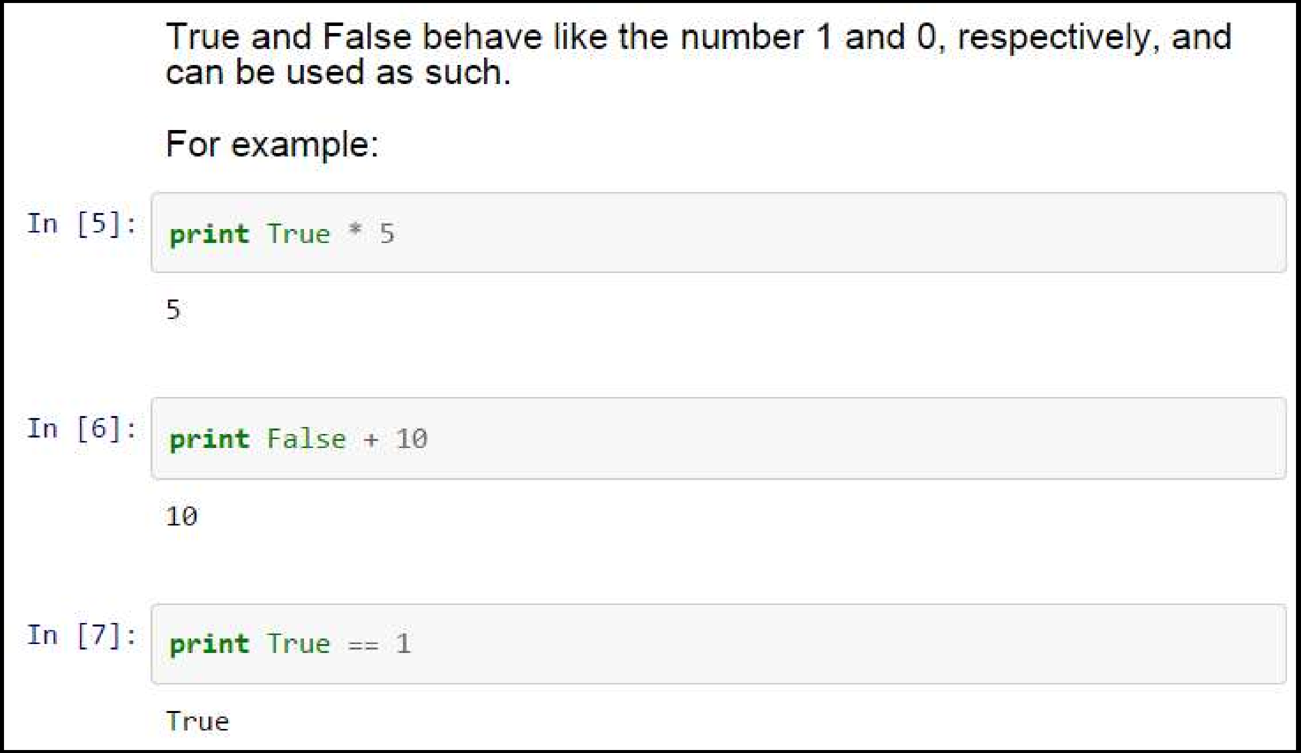
```
In [4]: print true
```

```
---------------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-4-a2a21570fba6> in <module>()
----> 1 print true

NameError: name 'true' is not defined
```





Booleans in Python 3

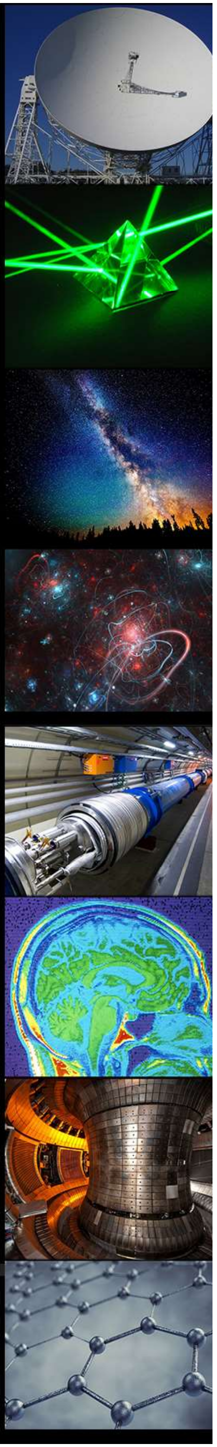


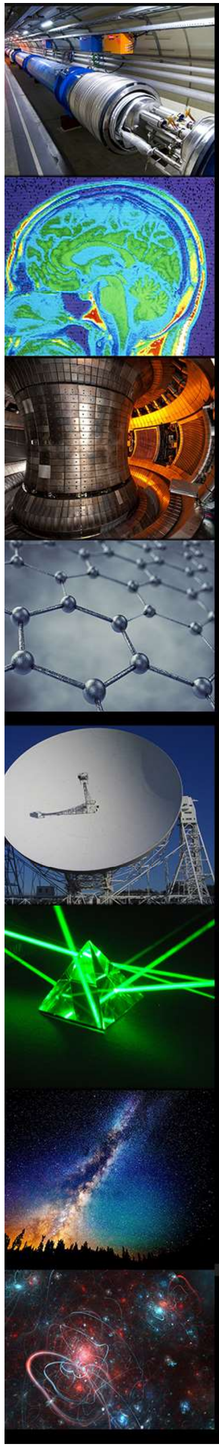
True and False behave like the number 1 and 0, respectively, and can be used as such.

For example:

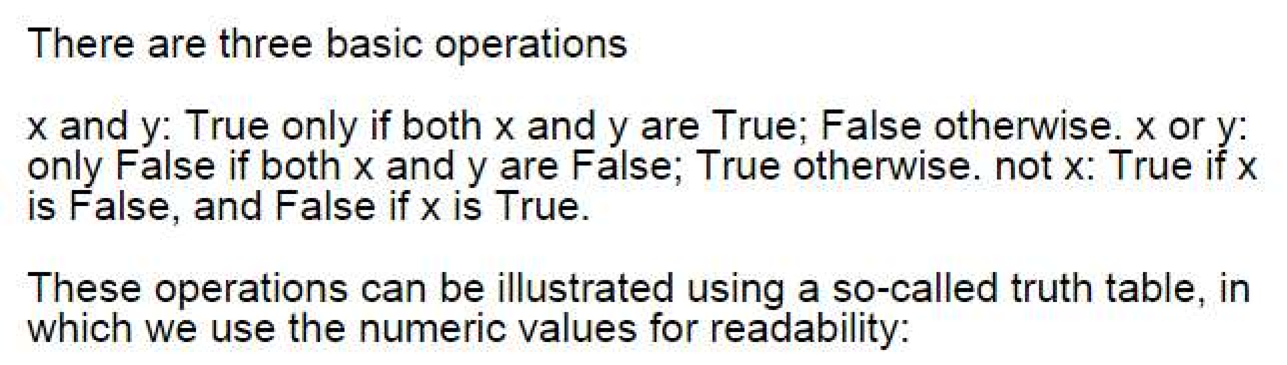
```
In [5]: print True * 5

        5

In [6]: print False + 10

        10

In [7]: print True == 1

        True
```

Though please don’t try doing maths with Booleans!





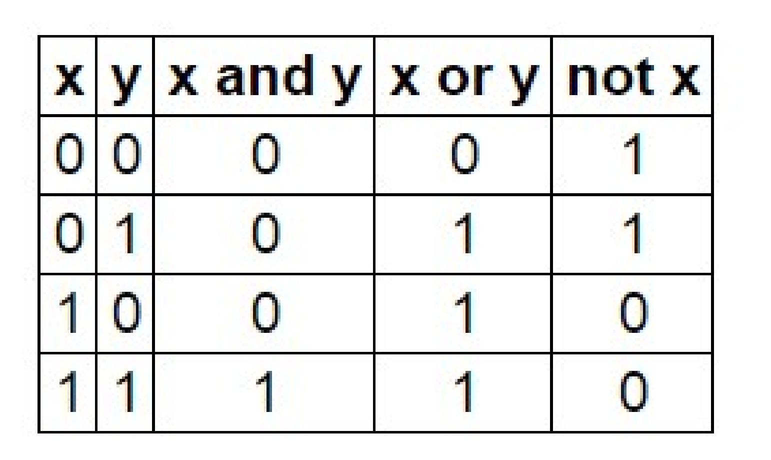
Basic Operations 1



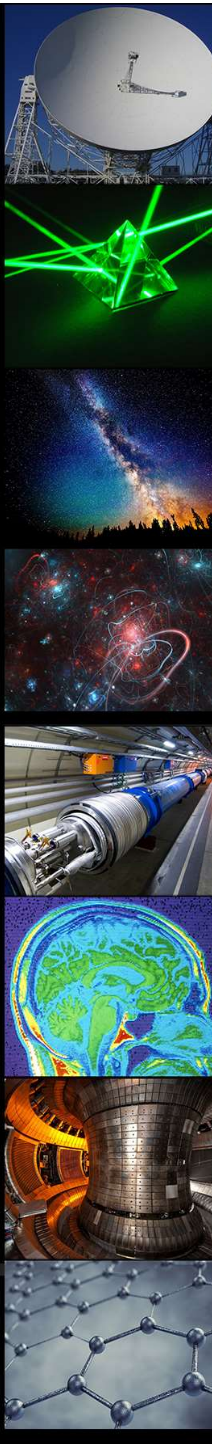
There are three basic operations

x and y: True only if both x and y are True; False otherwise. x or y: only False if both x and y are False; True otherwise. not x: True if x is False, and False if x is True.

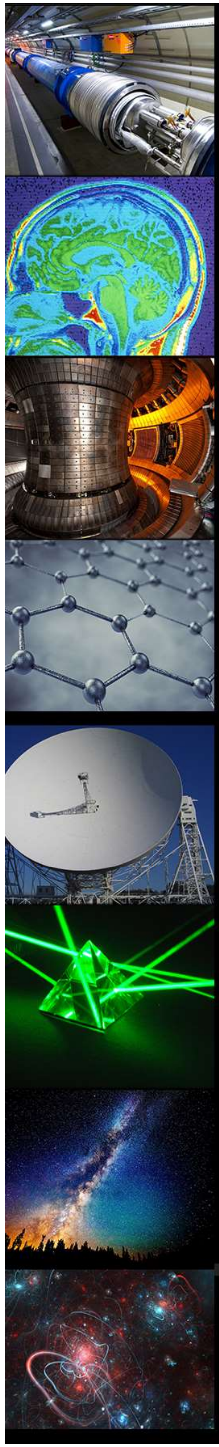
These operations can be illustrated using a so-called truth table, in which we use the numeric values for readability:

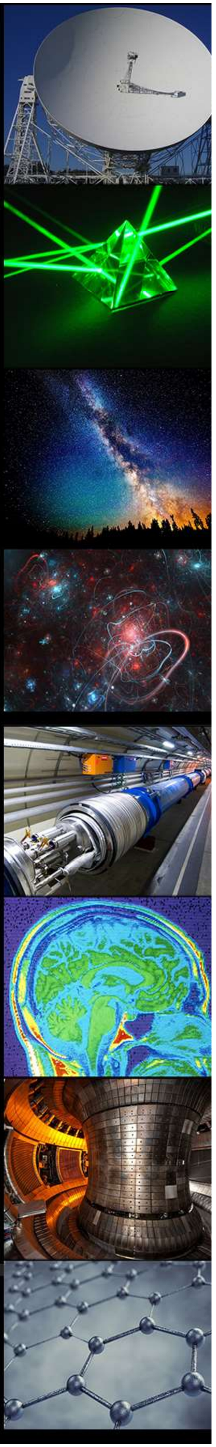


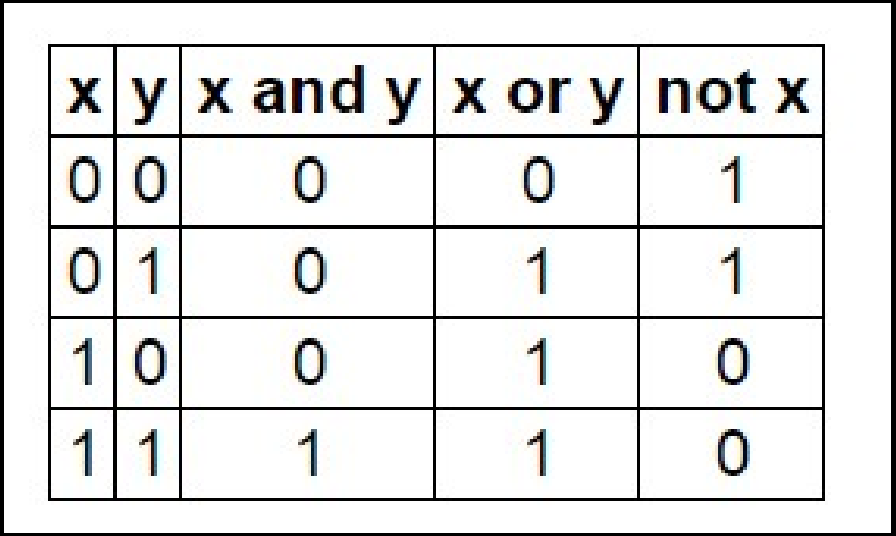
x	y	x and y	x or y	not x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0



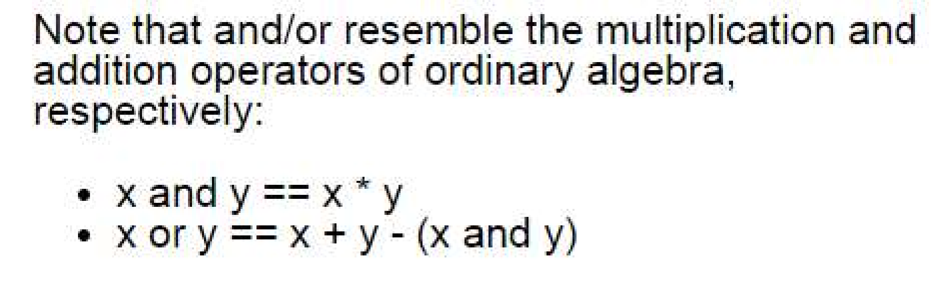
Basic Operations 2





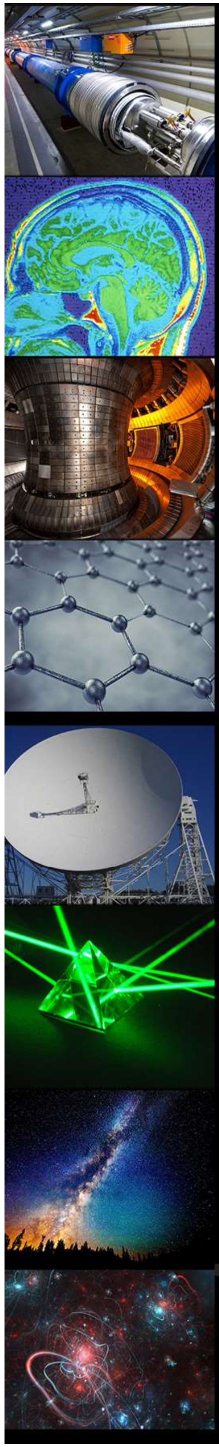


x	y	x and y	x or y	not x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

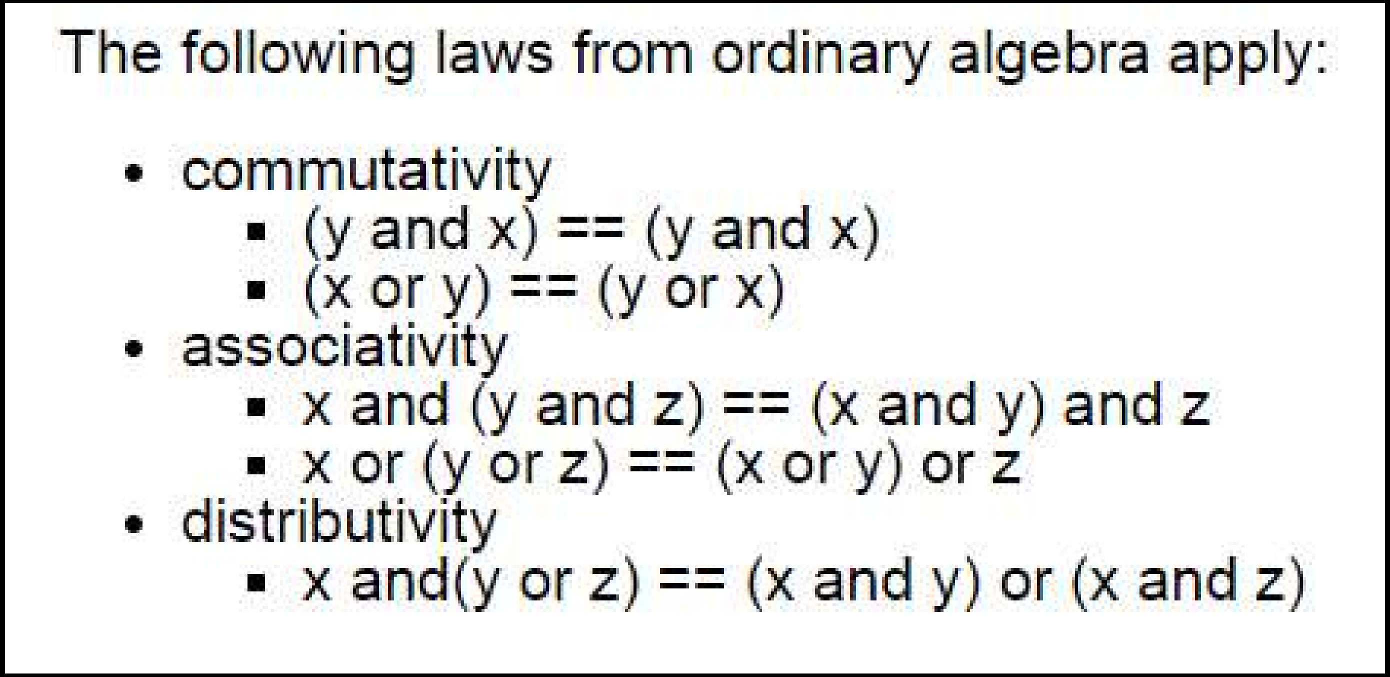


Note that and/or resemble the multiplication and addition operators of ordinary algebra, respectively:

- x and y == x * y
- x or y == x + y - (x and y)

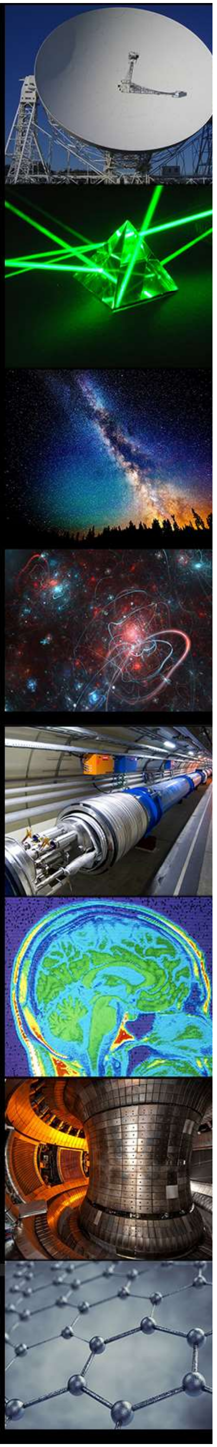


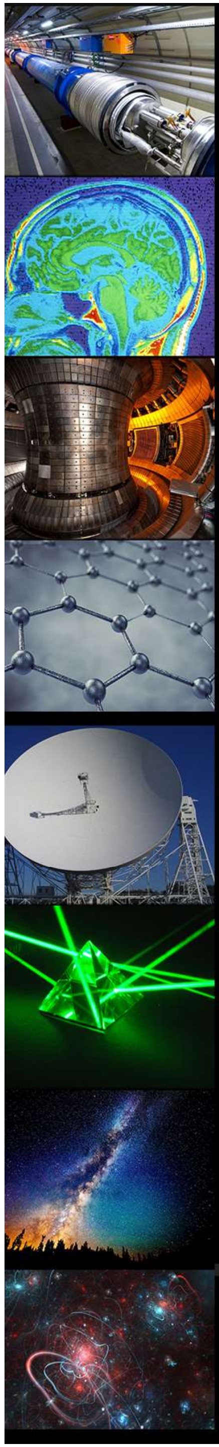
Laws



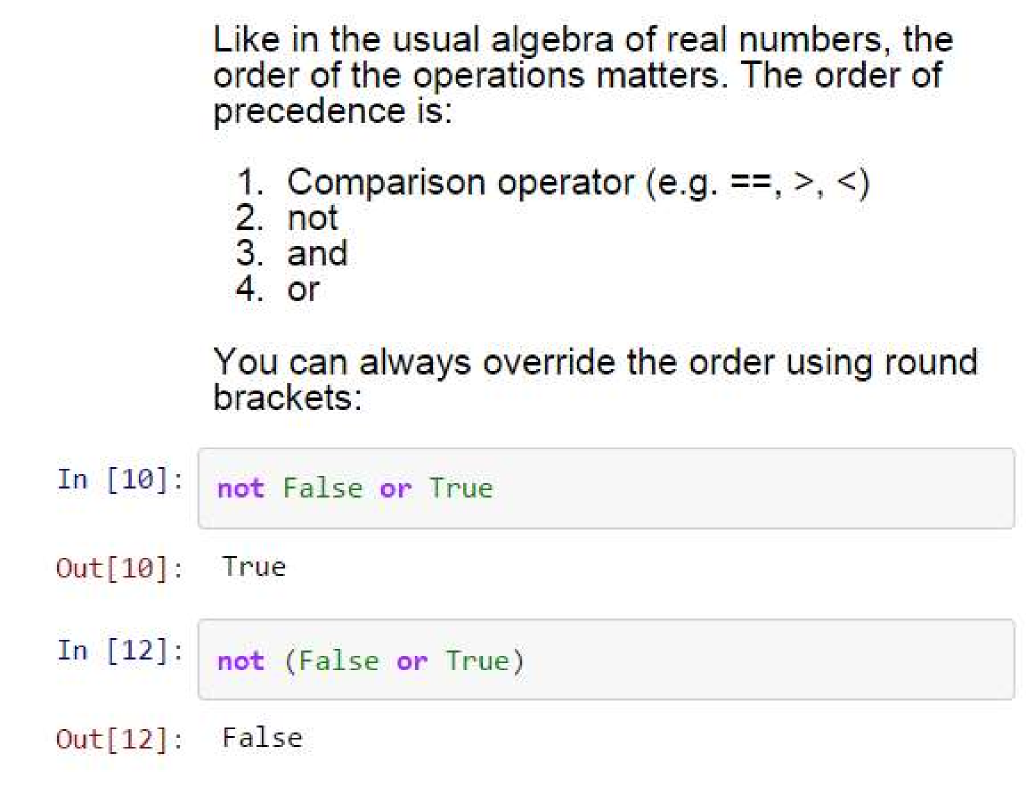
The following laws from ordinary algebra apply:

- commutativity
  - (y and x) == (y and x)
  - (x or y) == (y or x)
- associativity
  - x and (y and z) == (x and y) and z
  - x or (y or z) == (x or y) or z
- distributivity
  - x and(y or z) == (x and y) or (x and z)





Order of Operations 1



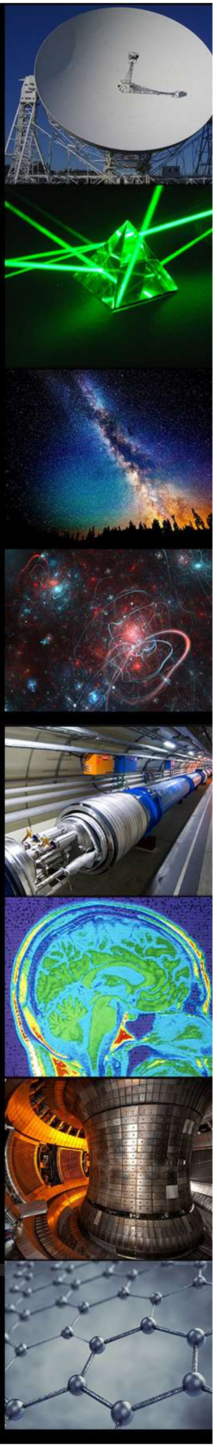
Like in the usual algebra of real numbers, the order of the operations matters. The order of precedence is:

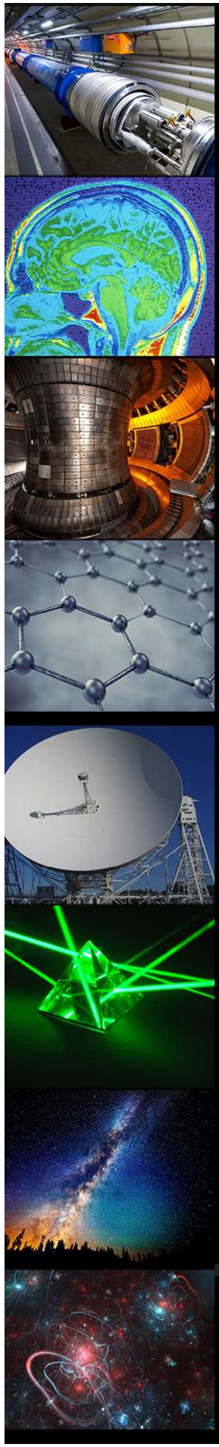
1. Comparison operator (e.g. ==, >, <)
2. not
3. and
4. or

You can always override the order using round brackets:

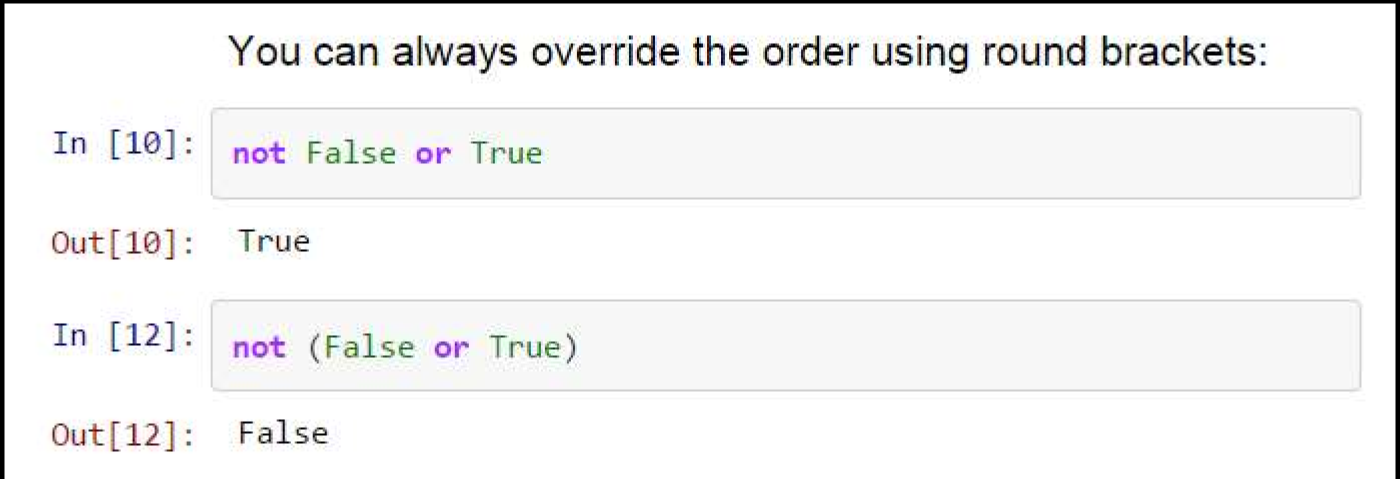
```
In [10]: not False or True

Out[10]: True

In [12]: not (False or True)

Out[12]: False
```

Recall from your maths: BODMASS





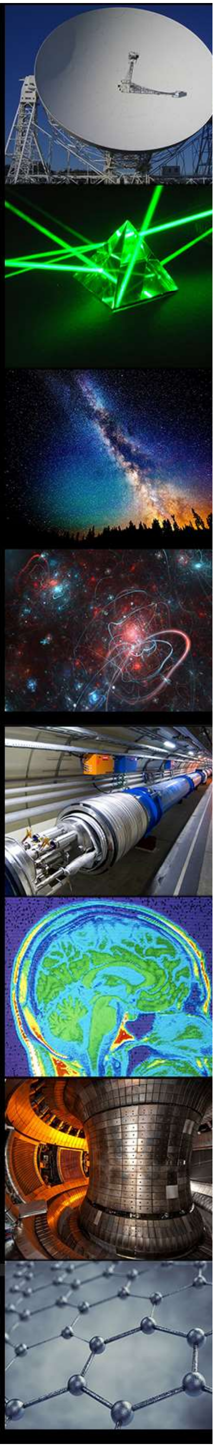
Order of Operations 2

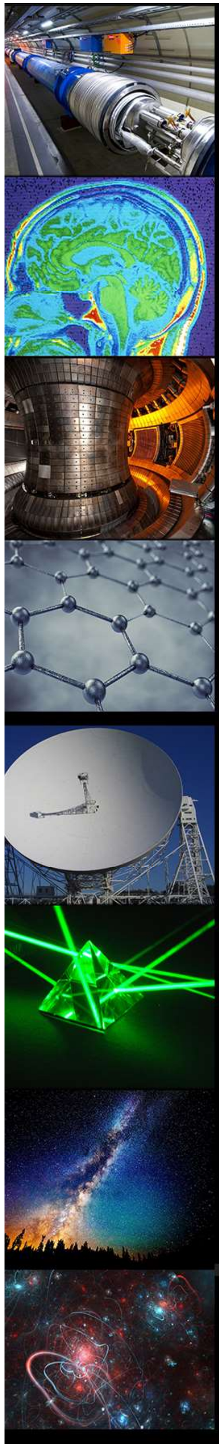


You can always override the order using round brackets:

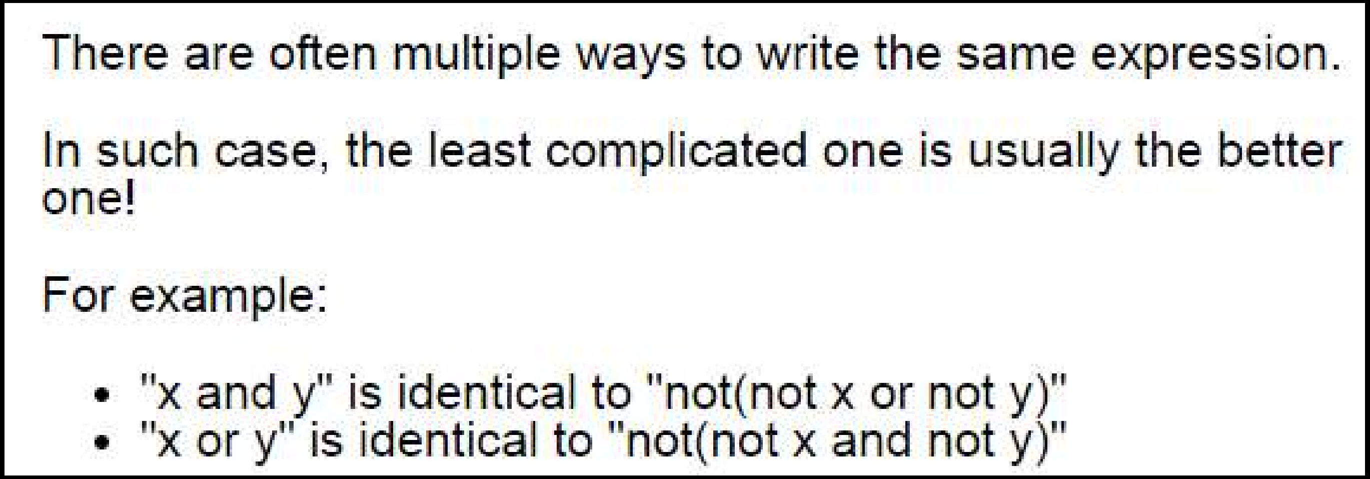
```
In [10]:  not False or True

Out[10]:  True

In [12]:  not (False or True)

Out[12]:  False
```

Recall from your maths: BODMASS





Rewriting Expressions

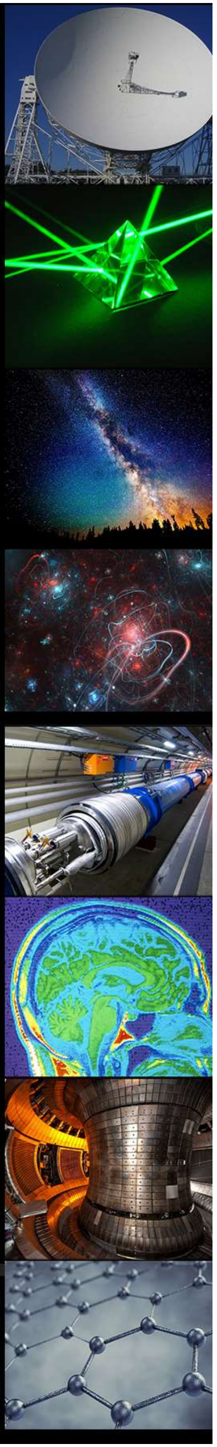


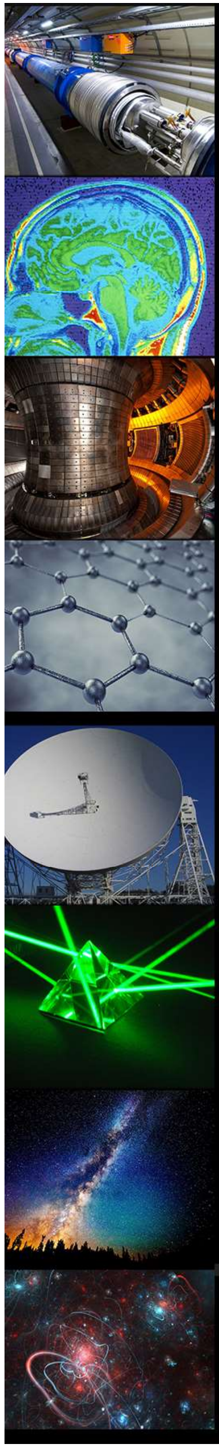
There are often multiple ways to write the same expression.

In such case, the least complicated one is usually the better one!

For example:

- "x and y" is identical to "not(not x or not y)"
- "x or y" is identical to "not(not x and not y)"



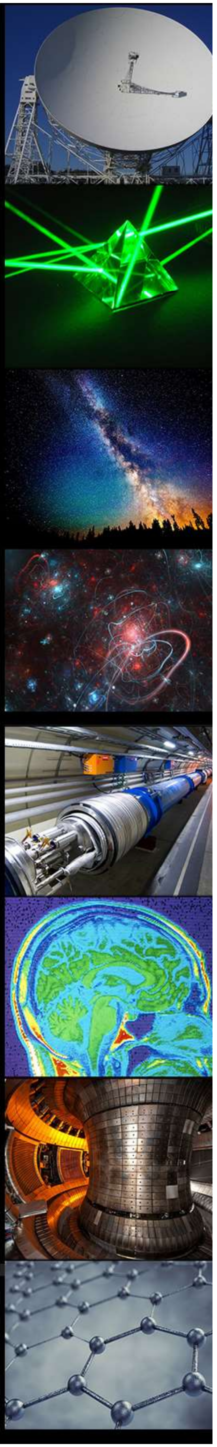


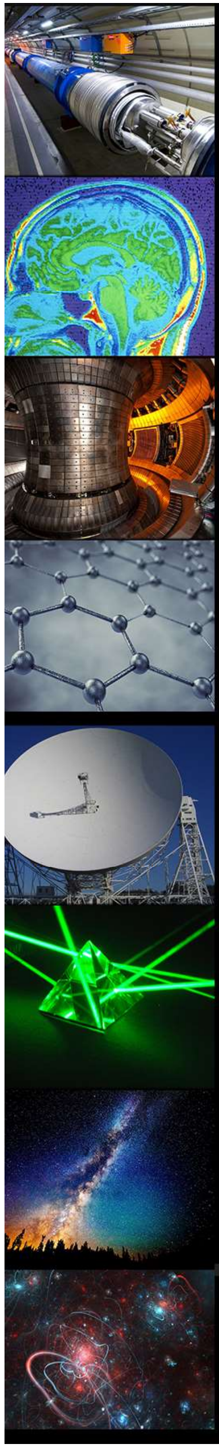
Set

An unordered collection, without duplicates (like Java).

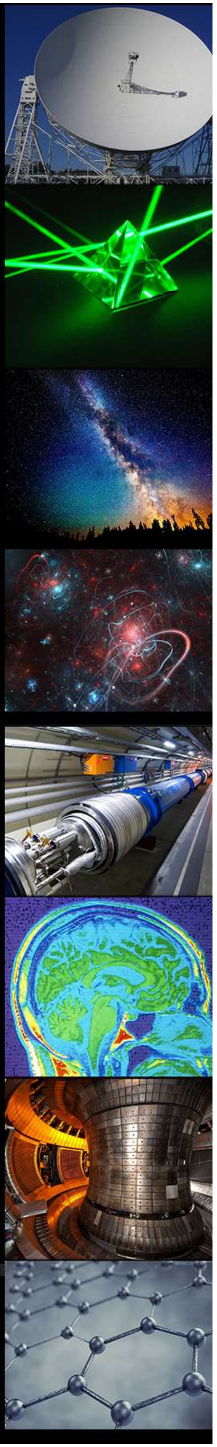
Syntax is like dictionary, but no " : " between key-value.

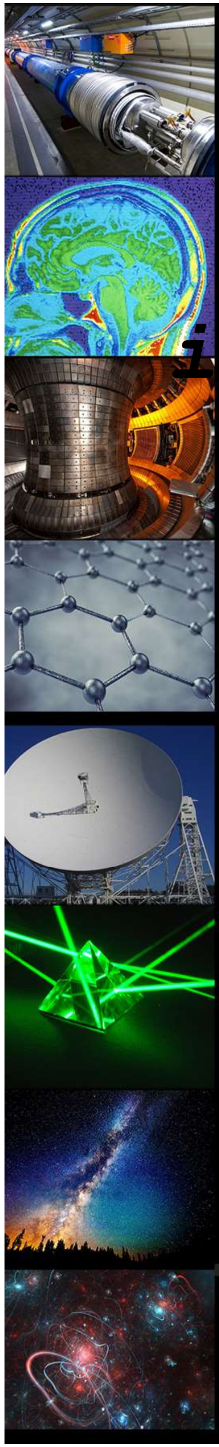
```
>>> aset = { 'a', 'b', 'c' }
>>> aset
{'a', 'c', 'b'}
>>> aset.add('c")                # no effect,
'c' already in set
>>> aset
{'a', 'c', 'b'}
```



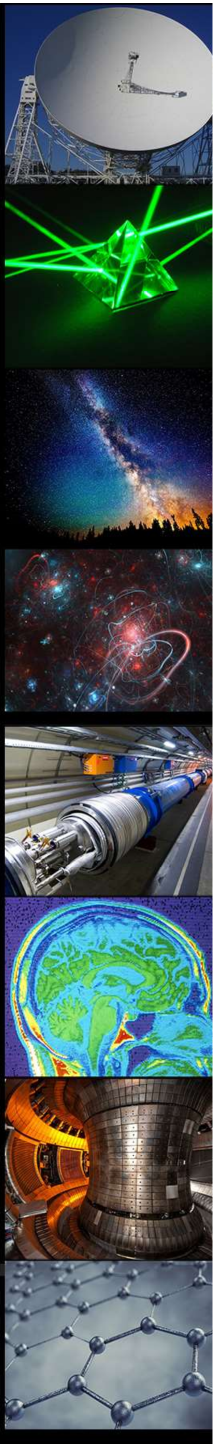


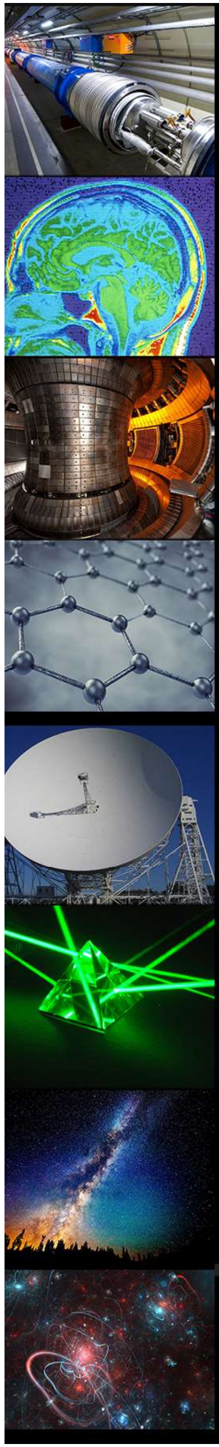
`set.discard('cat')`	remove cat. No error if not in set.
`set.remove('cat')`	remove cat. Error if not in set.
`set3 = set1.union(set2)`	doesn't change set1.
`set4 = set1.intersection(set2)`	doesn't change set1.
`set2.issubset( set1 )`	
`set2.issuperset( set1 )`	
`set1.difference( set2 )`	`element in set1 not set2`
`set1.symmetric_difference(set2)`	xor
`set1.clear( )`	remove everything





```
>>> aset = { 'a', 'b', 'c' }
>>> 'a' in aset
True
>>> 'A' in aset
False
```





```
if condition :
body
elif condition :
body
else:
body
```

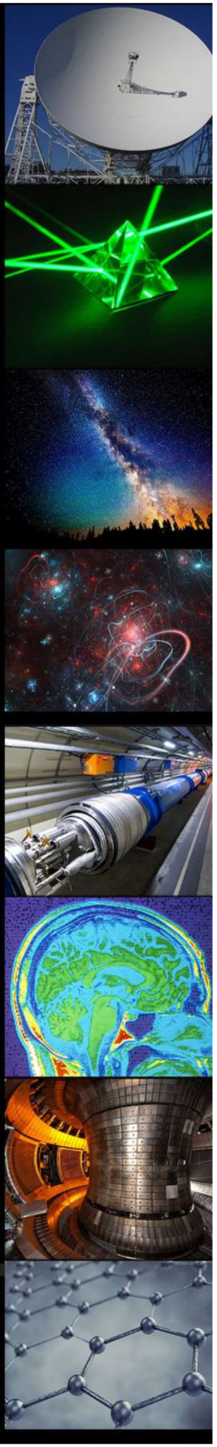
```
if x%2 == 0:
y = y + x
else:
y = y - x
```

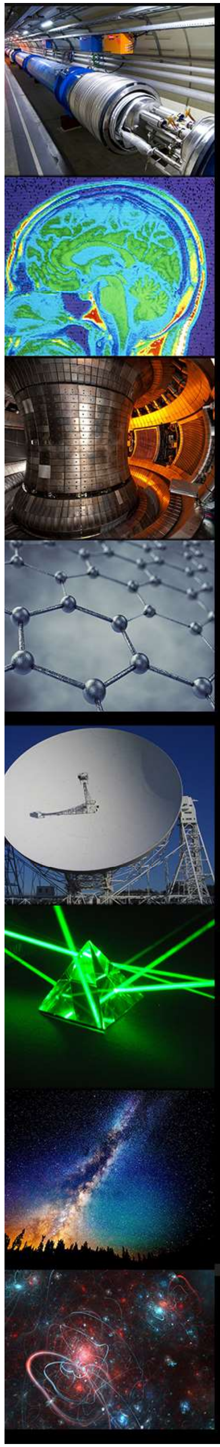
```
while condition:
body
```

```
while count <
10:
   count =
2*count
```

```
for name in
iterable:
body
```

```
for x in
[1,2,3]:
sum = sum + x
```





range: create a sequence

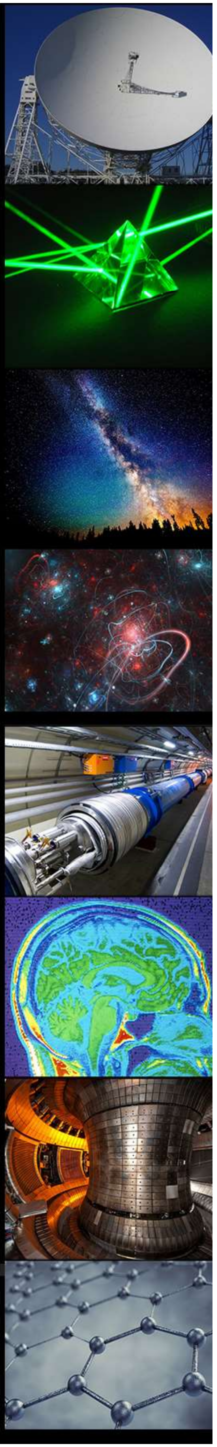
range([*start*,] *stop*[, *step*])

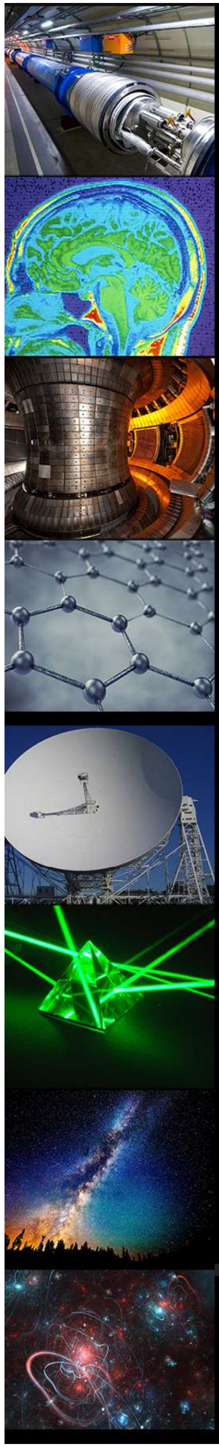
Generate a list of numbers from `start` to `stop` stepping every `step`

`start` defaults to 0, `step` defaults to 1

Example

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1, 9)
[1, 2, 3, 4, 5, 6, 7, 8]
>>> range(2, 20, 5)
[2, 7, 12, 17]
```

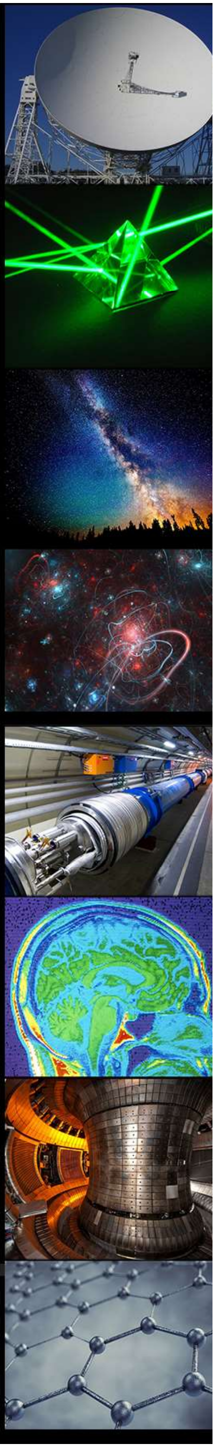


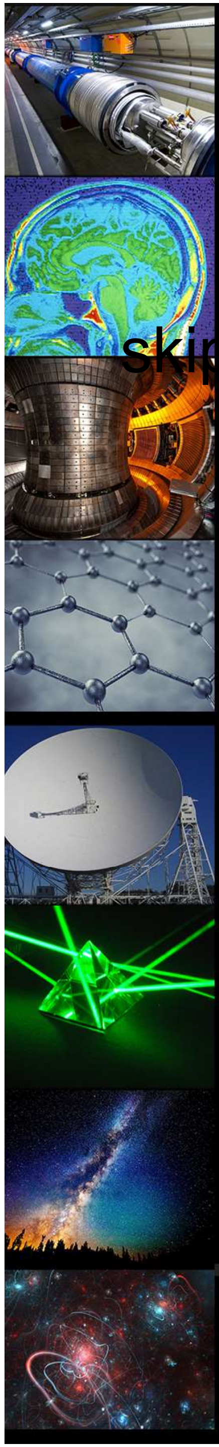


for loop using range( )

Use `range` to generate values to use in for loop

```
>>> for i in range(1,4):
     print (i)
1
2
3
```



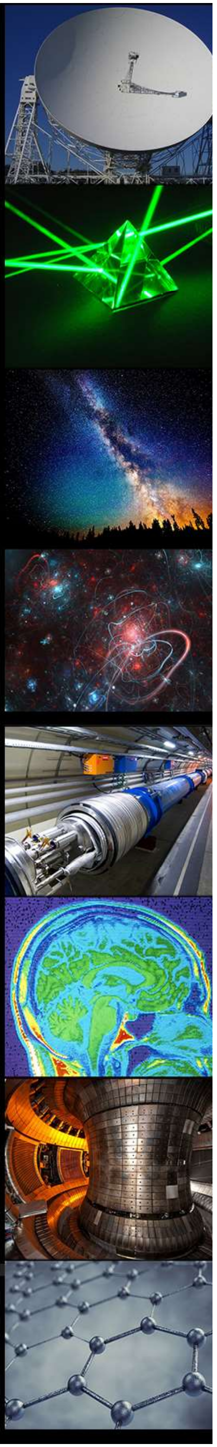


Skip

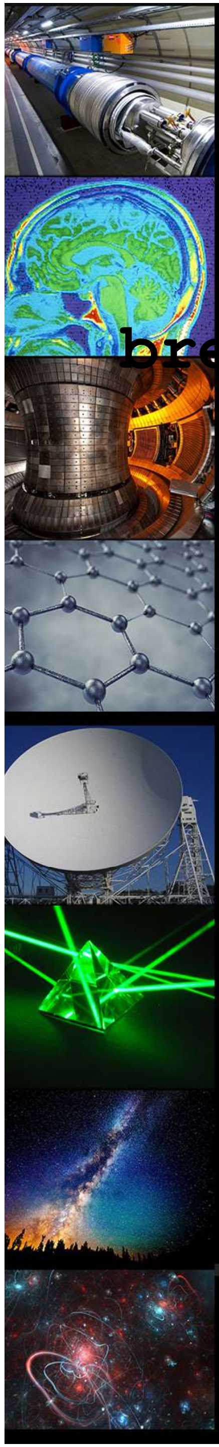
loop iteration using **continue**

```
for x in range(10):
      if x%2 == 0:
            continue
      print x
```

```
1
3
5
7
```



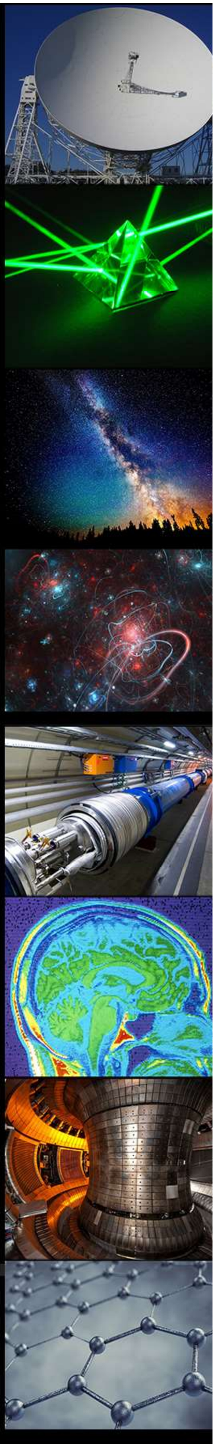
break

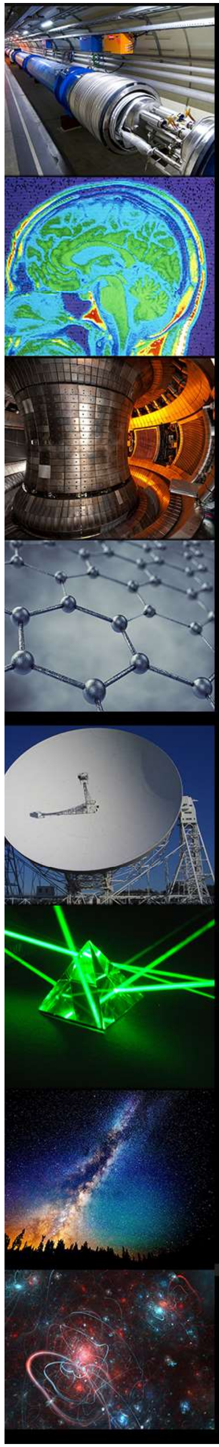


bre

```
for number in range(10):
    if number == 4:
         print 'Breaking'
         break
    else:
        print (number)
```

```
0
1
2
3
Breaking
```

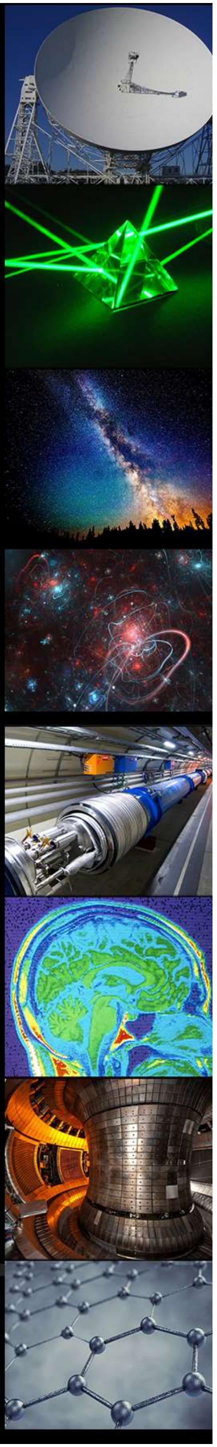


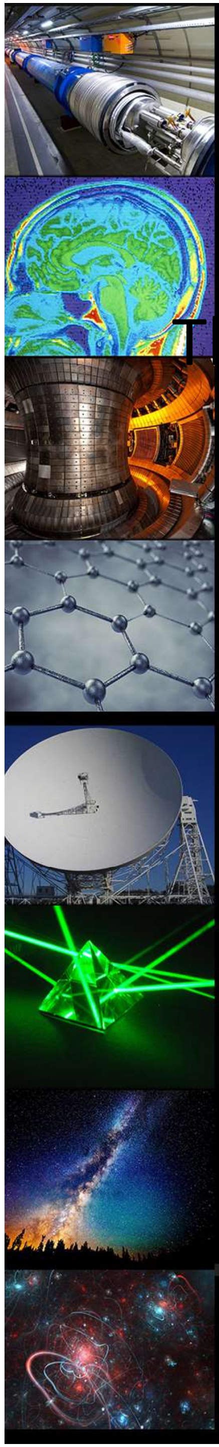


dir: show all methods & attributes

dir returns all methods for a class or object

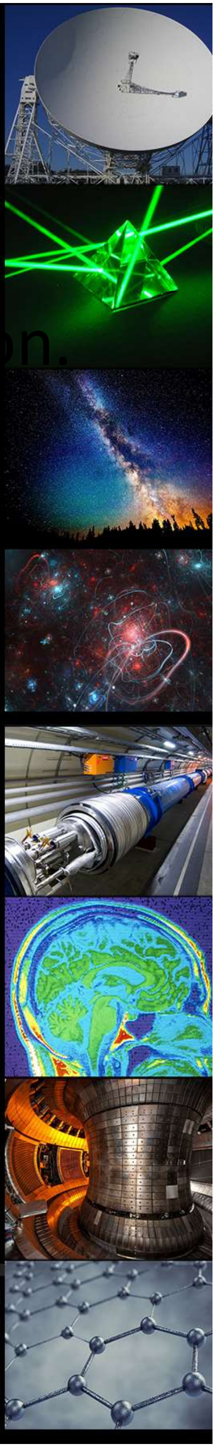
```
>>> lst = [1, 3, 2]

>>> dir(lst)

['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__',
'__getslice__',  ...

 '__setitem__', '__setslice__', '__str__', 'append',
'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort'


>>> dir( math )     # import math first

['__doc__', '__name__', '__package__', 'acos', 'asin",
'atan',
 'atan2', 'ceil', 'copysoign', 'cos', 'degrees', 'e',
'exp',
 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'hypot', ...
```

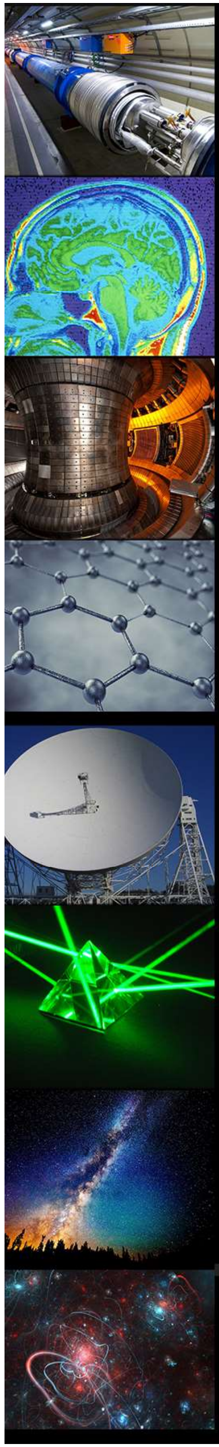




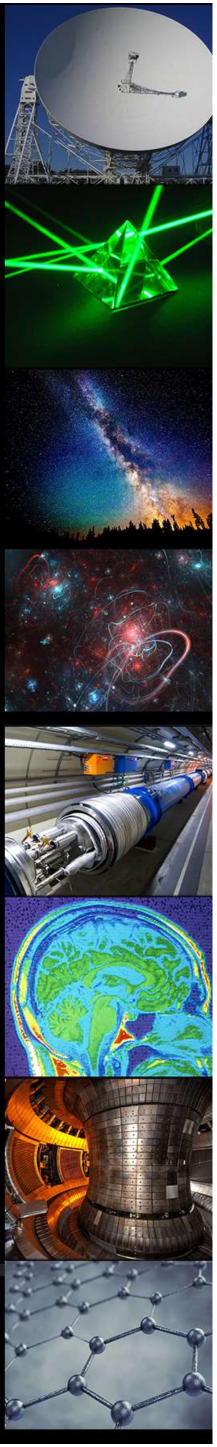
Getting Help

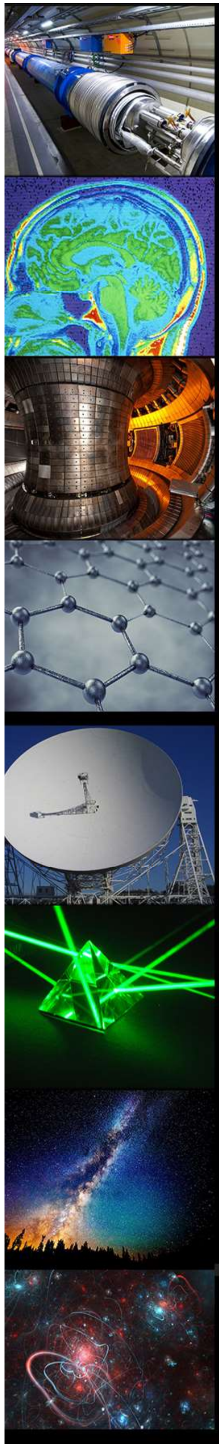
```
>>> help(str)
Help on class str in module __builtin__:

class str(basestring)
 |  str(object)-> string
 |
 |  Return a nice string representation of the
object.
 |
 |  Method resolution order:
 |      str
 |      basestring
 |      object
 |
 |  Methods defined here:
 |  ...
```





Functions





Defining Functions

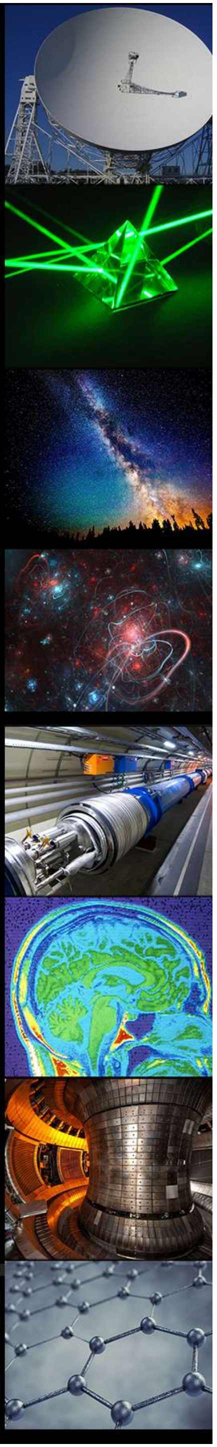
Syntax: **`def func(arg1, …):`**

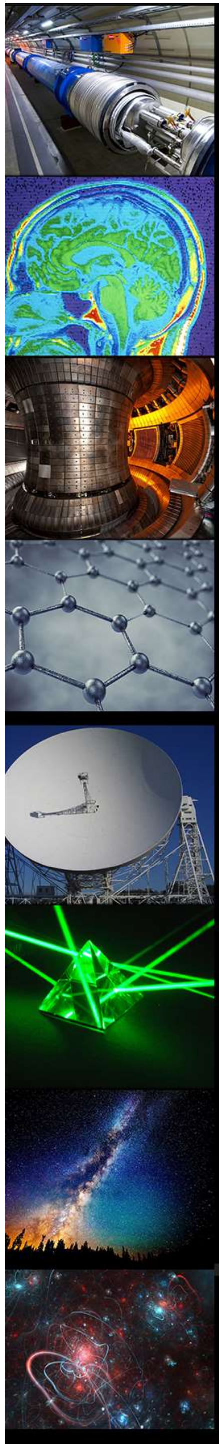
```
        body
```

Body of function must be indented

If no value is returned explicitly, function will return `None`

```
def average(num1, num2, num3):
    sum = num1 + num2 + num3
    avg = sum / 3.0
    return avg
```



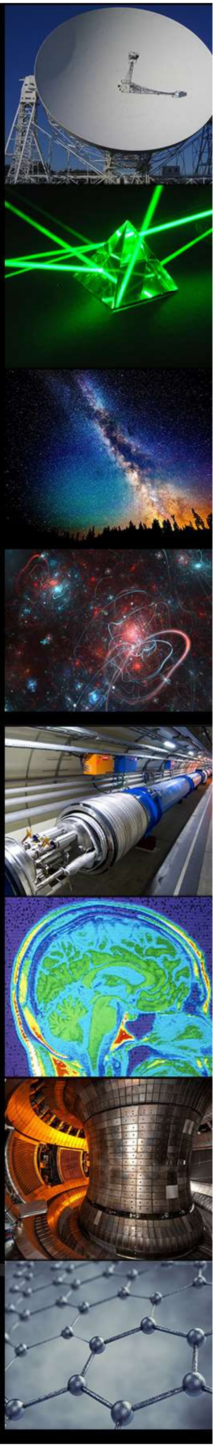


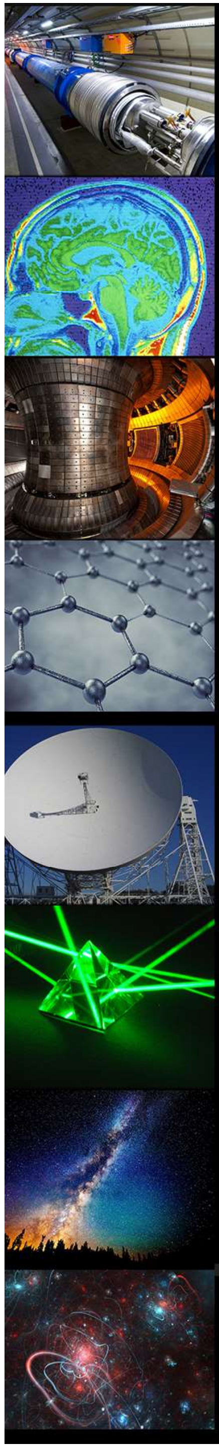
Function Parameters

Parameters can be any type

A function can take any number of parameters or none at all

```
def usage(programName, version):

      print("%s Version %i" % (programName,
version))

      print("Usage: %s arg1 arg2" %
programName)


>>> usage('Test', 1.0)

Test Version 1.0

Usage: Test arg1 arg2
```





Function Default Parameter values

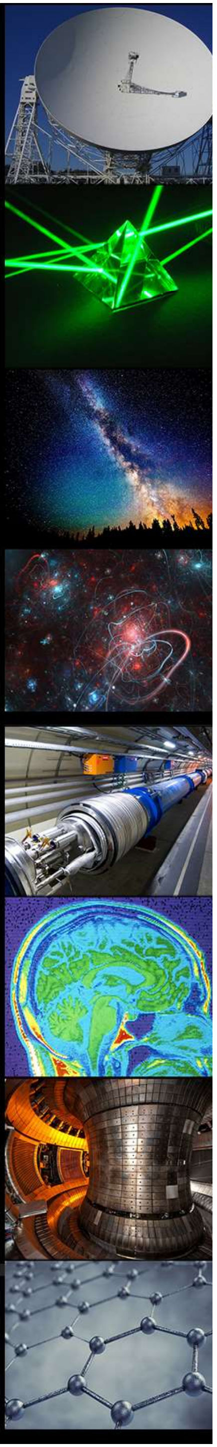
Parameters can be given a default values

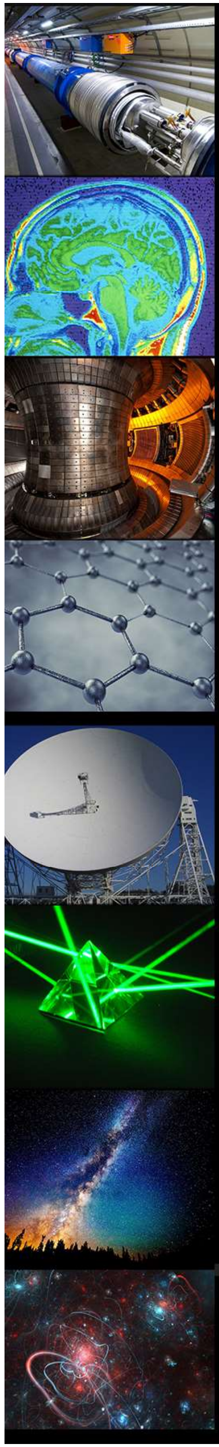
```
split(string, substr=' ')
```

The function can be called with fewer arguments than there are parameters

Parameters with default values must come last

```
>>> def printName(last, first, mi=""):
            print("%s, %s %s" % (last,
first, mi))

>>> printName("Smith", "John")
Smith, John
>>> printName("Smith", "John", "Q")
Smith, John Q
```





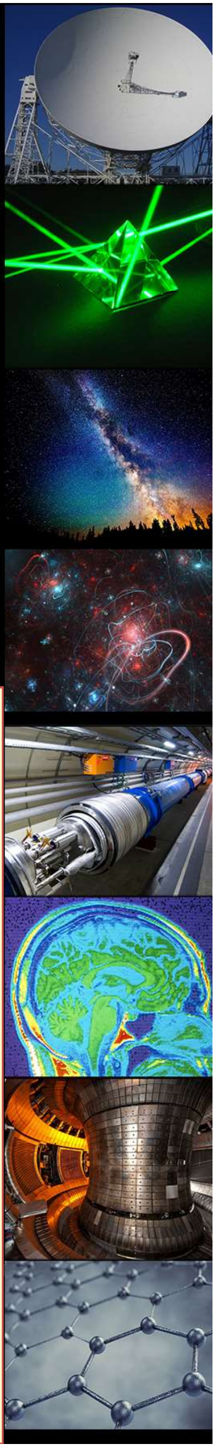
Keyword Arguments

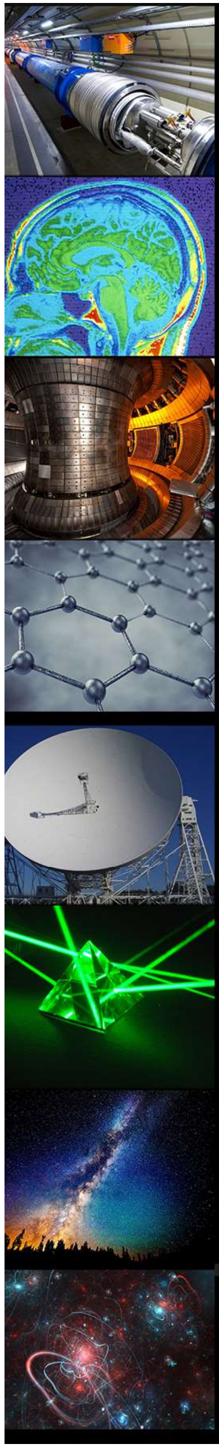
Functions can be invoked using the name of the parameter and a value

```
func(param=value, ...)
```

- The order of values passed by keyword does not matter

```
def fun(key1="X", key2="X", key3="X",
key4="X"):
      '''function with keywords and default
values'''
      print(key1, key2, key3, key4)

>>> fun(key3="O", key2="O")
X O O X
>>> fun(key4='Z')
X X X Z
```

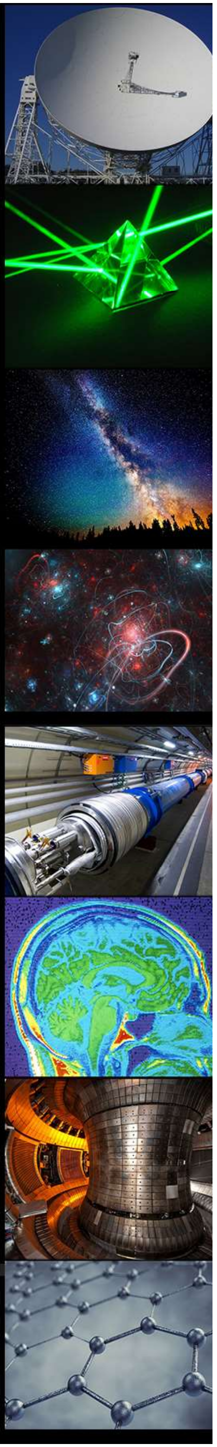


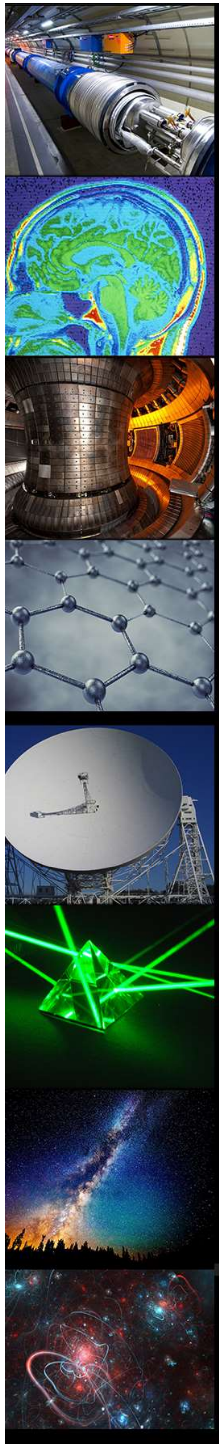


Functions at Values

- Functions can be used just like any other data type
- Functions can be assigned to variables

```
def sub(a, b):
    return a-b

>>> op = sub
>>> print op(3, 5)
-2
>>> type(op)
<type 'function'>
```

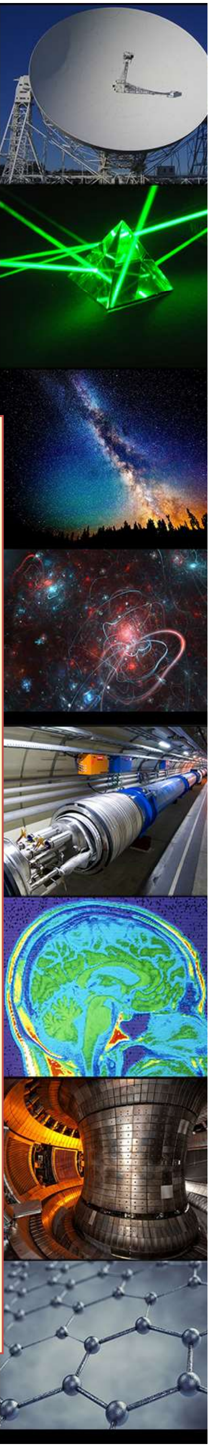


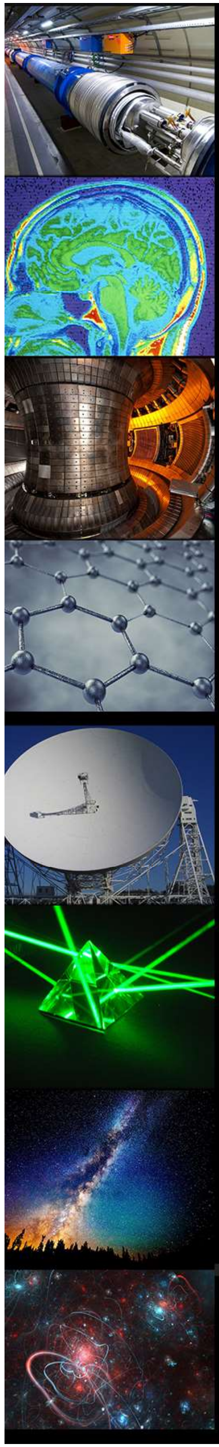


Functions as Parameters

Functions can be passed to other functions

```
def convert(data, convertFunc):
    for i in range(len(data)):
        data[i] = convertFunc(data[i])
    return data

>>> convert(['1', '5', '10', '53'], int)
[1, 5, 10, 53]
>>> convert(['1', 'nerd', '10', 'hi!'], len)
[1.0, 5.0, 10.0, 53.0]
>>> convert(['1', '5', '10', '53'], complex)
[(1+0j), (5+0j), (10+0j), (53+0j)]
```

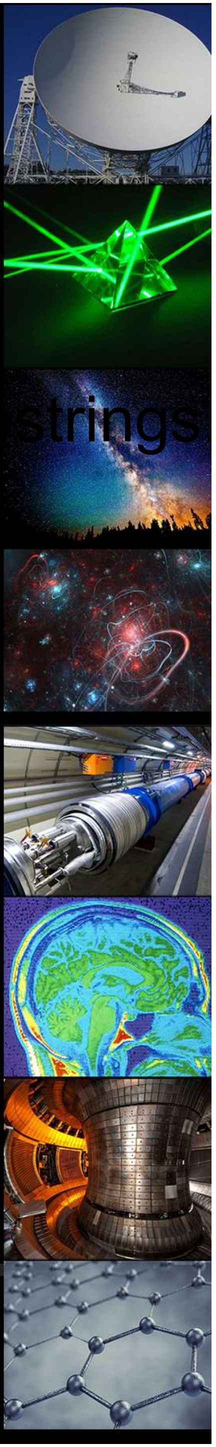


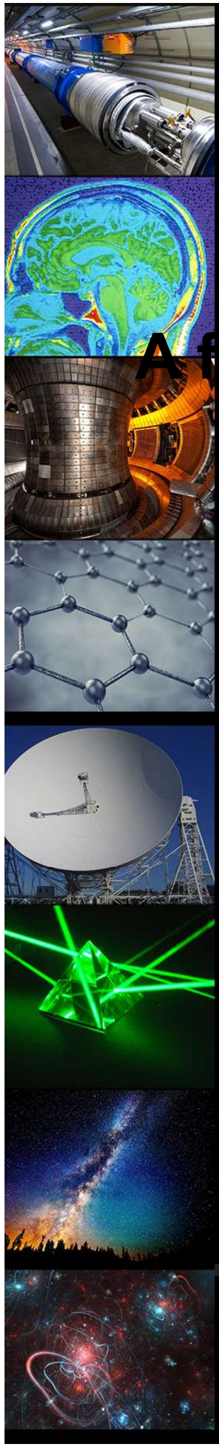


Functions can return multiple values

Return a tuple of values.

```
def separate(text, size=3):
      head = text[:size]
      tail = text[size:]
      return (head,tail)
      # ok to omit parens: start,last =
separate(...)
(start,last) = separate('GOODBYE', 4)
start
GOOD
last
BYE
```





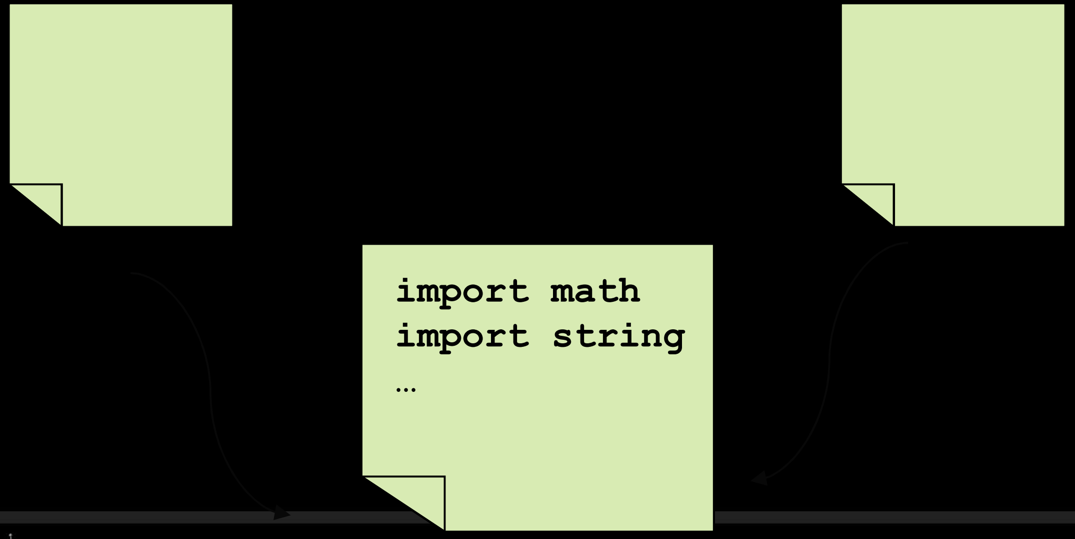
Modules

A f

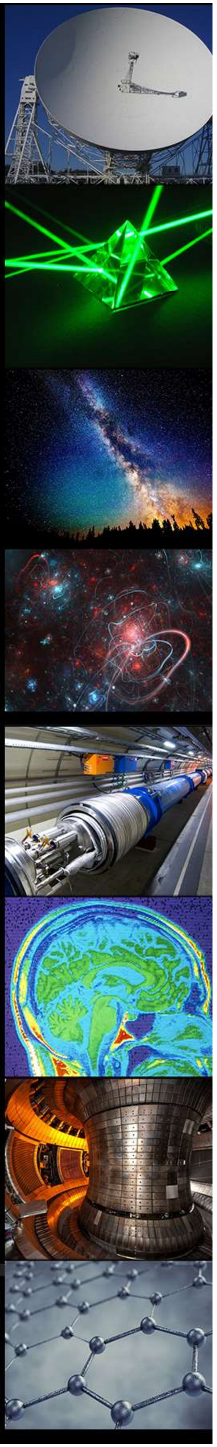
Modules can be “imported”

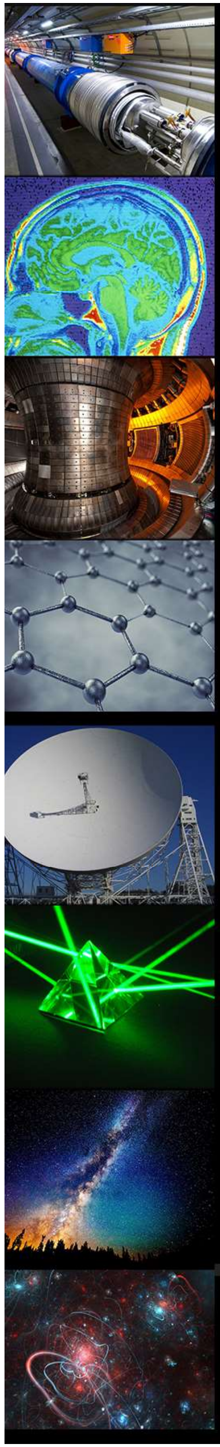
Module file name must end in .py

Used to divide code between files



```
import math
import string
…
```



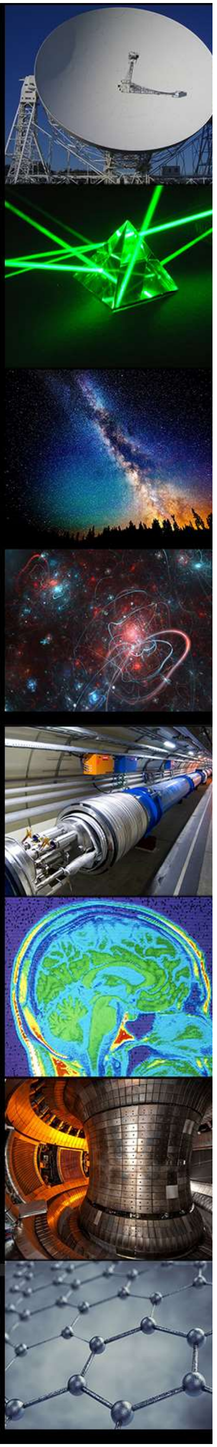


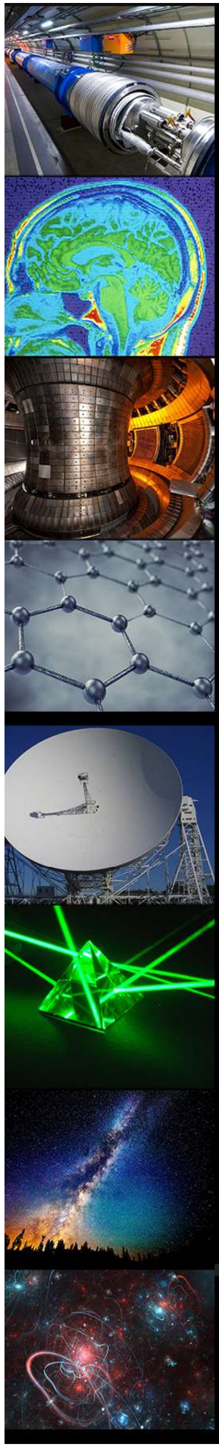
import Statement

import <module name>

- **module name** is the file name without **.py** extension
- You must use the module name to call functions

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin',
'atan', 'atan2', 'ceil', 'cos', 'cosh', 'e',
'exp', 'fabs', 'floor', 'fmod', 'frexp',
...]
>>> math.e
2.71828182846
>>> math.sqrt(2.3)
1.51657508881
```





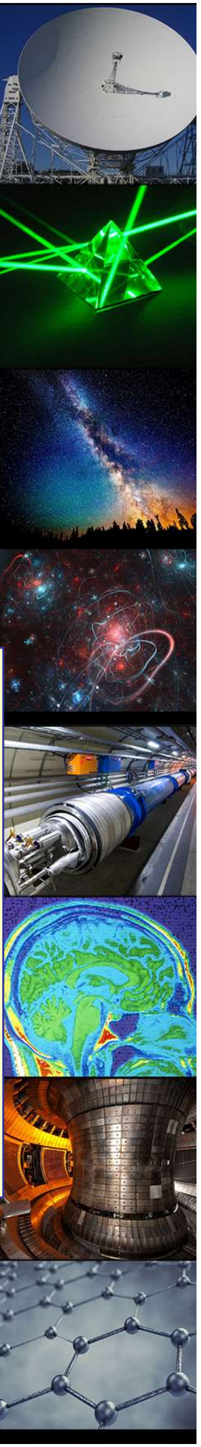
import specific names

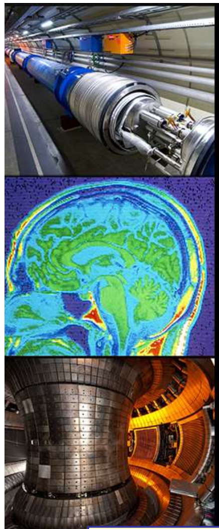
from <module> import <name>

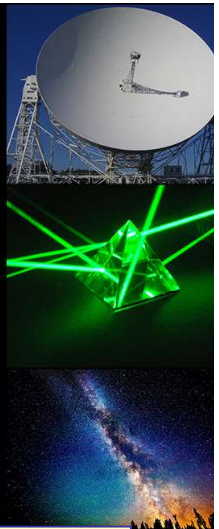
Import a specific name from a module into global namespace

Module name is not required to access imported name(s)

```
>>> from math import sqrt
>>> sqrt(16)
4
>>> dir(math)
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'math' is not defined
```





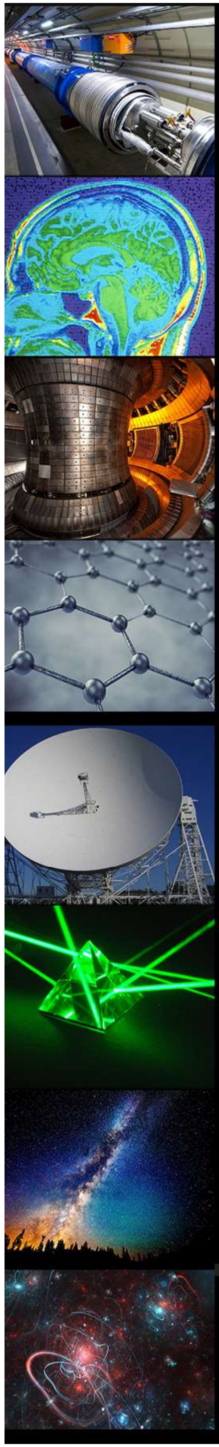


`import` all names from module

`from <module> import *`

Import everything into global namespace

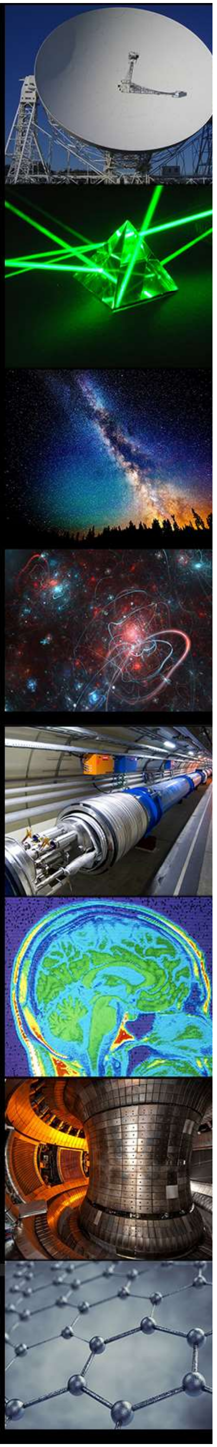
```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> from time import *
>>> dir()
['__builtins__', '__doc__', '__name__',
'accept2dyear', 'altzone', 'asctime', 'clock',
'ctime', 'daylight',    'gmtime', 'localtime',
'mktime', 'sleep', 'strftime', 'time', ... ]
>>> time()
1054004638.75
```



Python Standard Libraries

`sys`	System-specific parameters and functions
`time`	Time access and conversions
`thread`	Multiple threads of control
`re`	Regular expression operations
`email`	Email and MIME handling
`httplib`	HTTP protocol client
`tkinter`	GUI package based on TCL/Tk (in Python 2.x this is named Tkinter)

http://docs.python.org/library/index.html

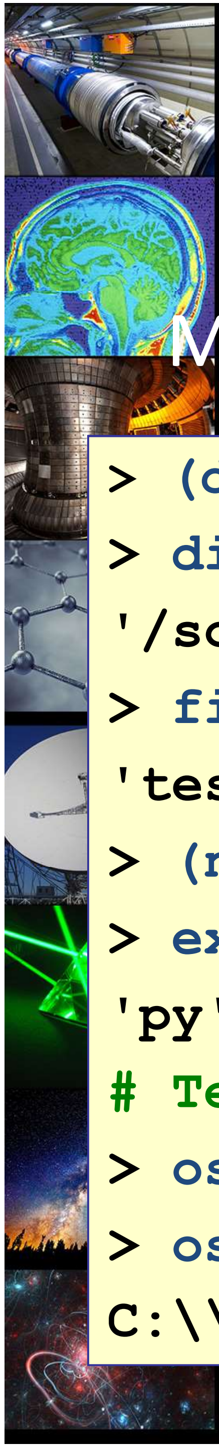


The `os` Module

Operating System related functions

```
import os
```

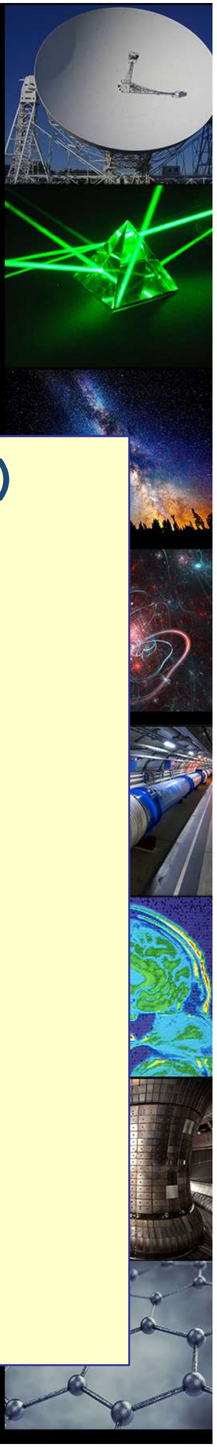
`os.getcwd( )`	Get current working directory
`os.chdir("/temp/somedir")`	Change dir. Use forward slash on MS Windows.
`os.getenv('JAVA_HOME')`	Get environment variable.
`os.unlink('filename')`	Remove regular file
`os.removedirs('dirname')`	Remove directory(s)
`stats = os.stat('file')`	Get file metadata: ctime, mtime, atime, uid, gid

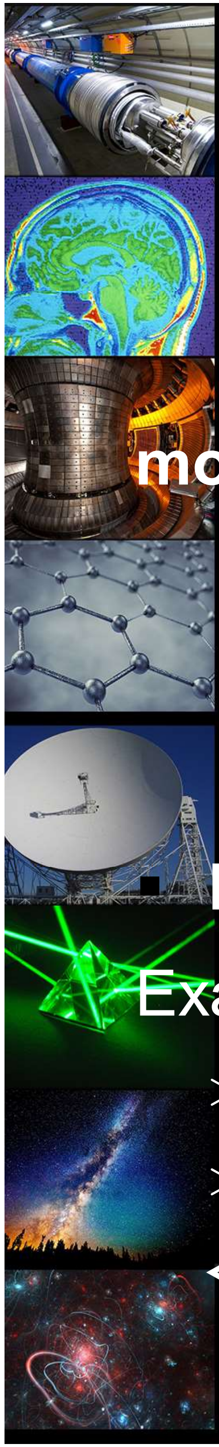


os.path

Module for manipulating paths

```
> (dir, file) = os.path.split("/some/path/test.py")
> dir
'/some/path'
> file
'test.py'
> (name,ext) = os.path.splitext(file)
> ext
'py'
# Test absolute path to file
> os.chdir('/Temp/examples')
> os.path.realpath('readme.txt')
C:\\Temp\\examples\\readme.txt
```





File Objects - `open` a file

```
open(filename, mode)
```

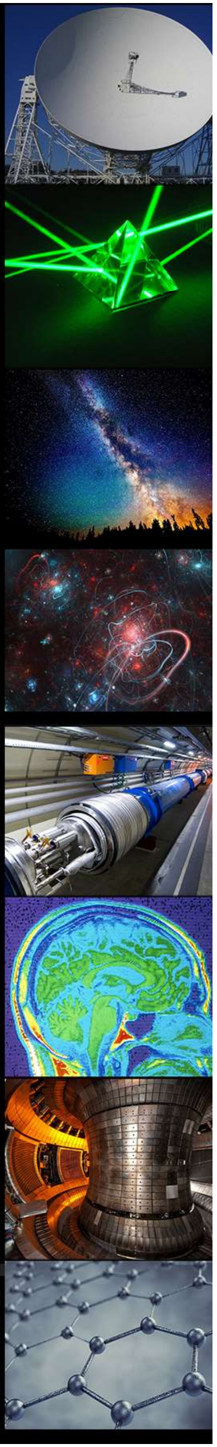
mode is one of:

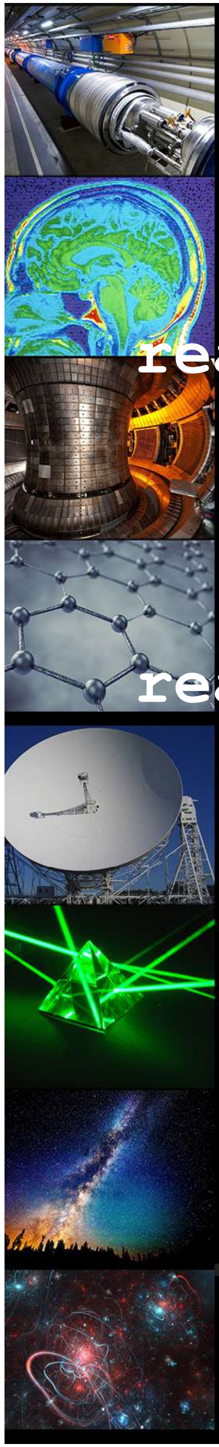
- `'r'` : Read
- `'w'` : Write
- `'a'` : Append

If a file opened for 'w' does not exist it will be created

Example

```
>>> inFile = open('input.txt', 'r')
>>> type(infile)
<type 'file'>
```





File Methods

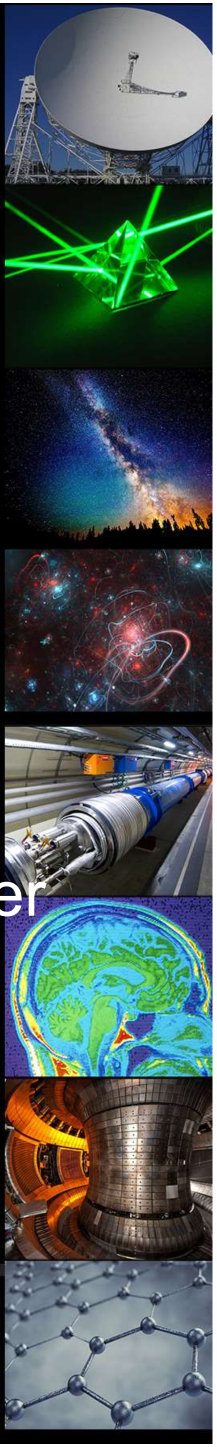
`read([size])`

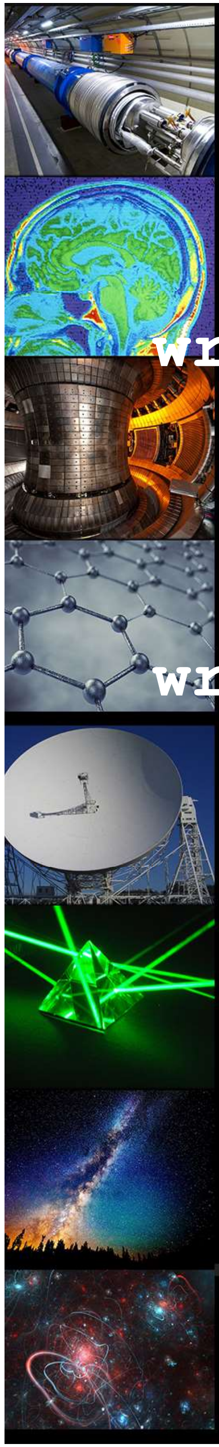
Read at most `size` bytes and return as a string

`readlines([size])`

Read the lines of the file into a list of strings.

Use `size` as an approximate bound on the number of bytes returned





File Methods for output

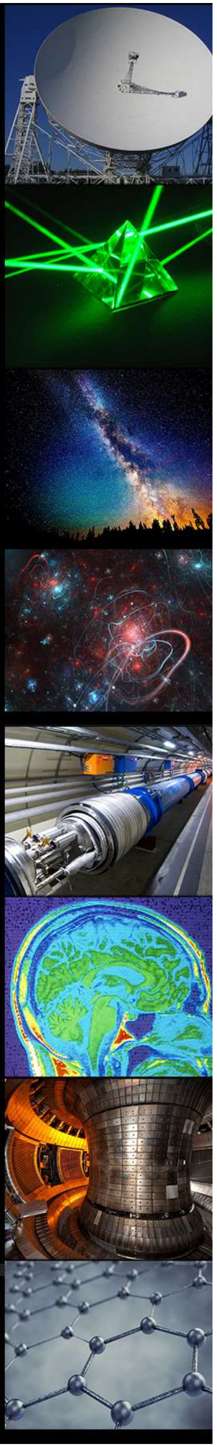
`write(text)`

Write `text` to the file

`writelines(string_sequence)`

Write each string in the sequence to the file

New lines are not added to the end of the strings



with to define a block for open

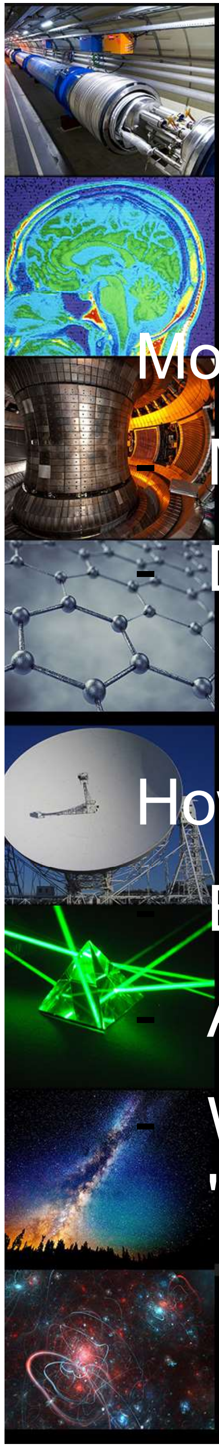
if an exception occurs.

```
with open("students.csv",'r') as student_file:
      # read each line of file
      for line in student_file:
            (id,name,email) = split(',', 3)
            print("%s has id %s" % (name,id))
```

Exceptions

Example: division by zero

```
>>> 1 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in ?
    1 / 0
ZeroDivisionError: integer division or modulo by
zero
```



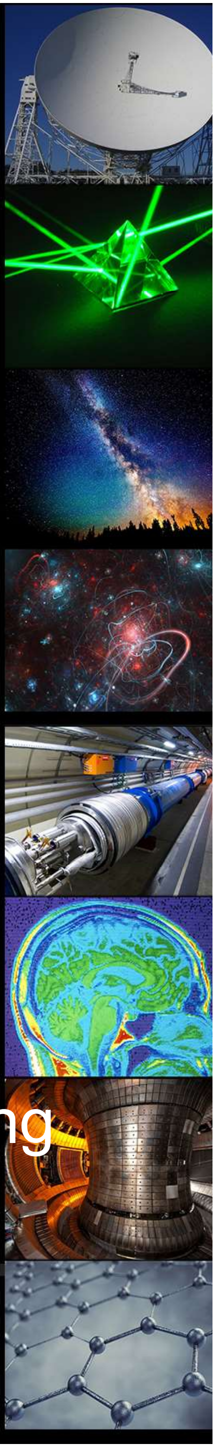
Exceptions

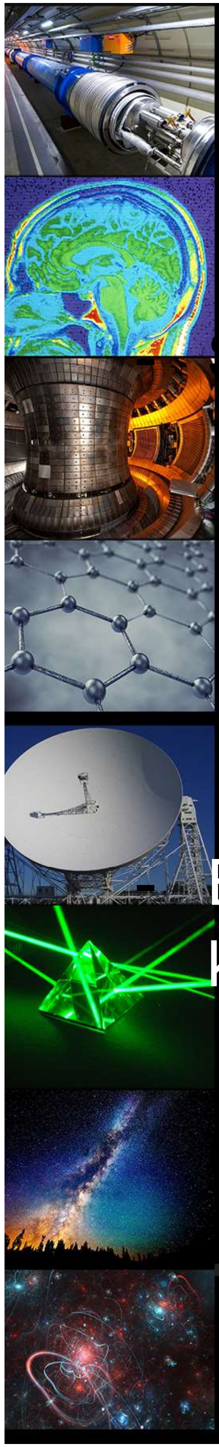
Motivation

- Move error handling code away from main code
- Deal with “exceptional” cases separately

How it works

- Exceptions are *thrown* (or raised) and *caught*
- An exception is caught by a catch code block
- When exception occurs, control jumps to a surrounding "catch" block. May *propagate* from function to caller.





Throwing Exceptions

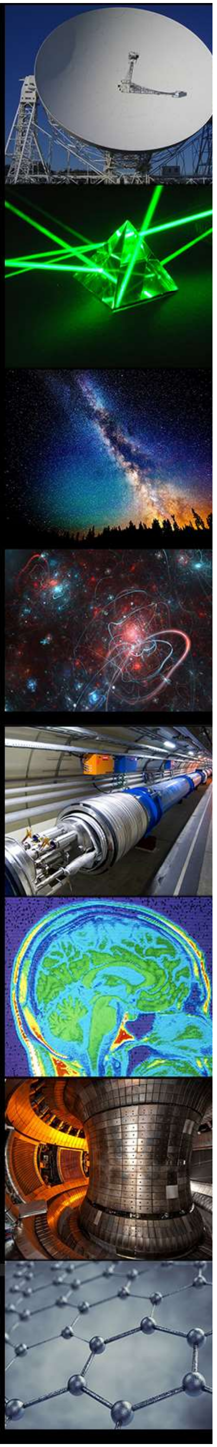
List index out of bounds

Invalid type conversions

Invalid arithmetic operation, e.g. math.sqrt(-1)

Exceptions can be thrown manually using the `raise` keyword

```
>>> raise ValueError, "Bad Value“
```

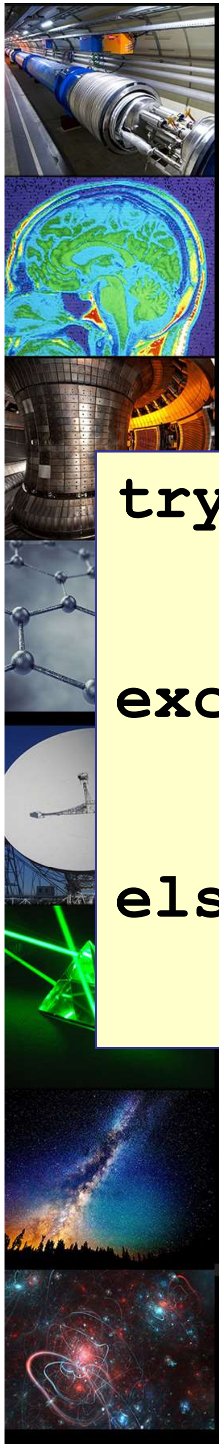


Catching an Exception

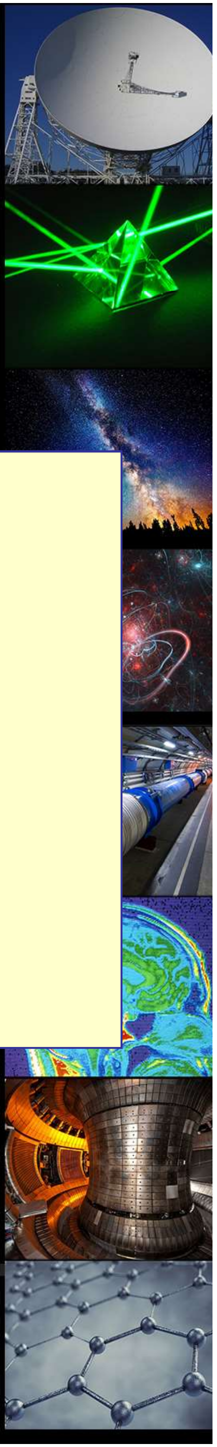
If a

```
try
    <code block>
except <Exception List1>:
    <exception handling code block>
except <Exception List2>:
    <exception handling code block>
except:
    <exception handler for all other exceptions>
else:
    <code to execute if no exception occurs>
```

Exception Example



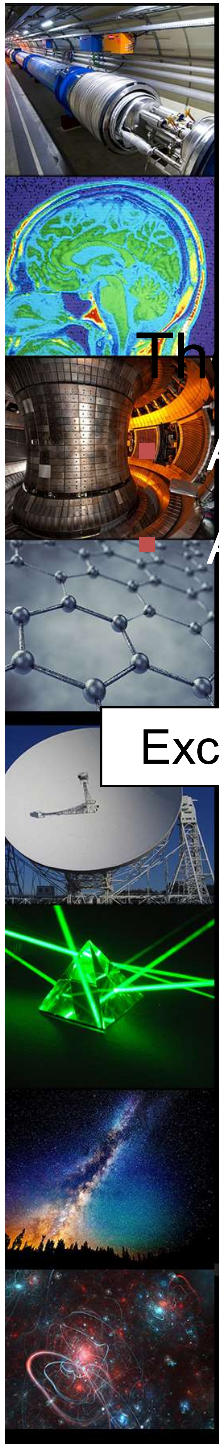
```
try:
    x = 1 / 0
except ZeroDivisionError:
    print('Division by zero')
else:
    print("x =", x)
```



First matching except block is used

```
try:
    x = 1 / 0
except IOError:
    print 'Input/Output error'
except:
    print 'Unknown error'
```

```
Unknown error
```



Types of Exceptions

Th a hierarchy of exceptions

- All built-in exceptions are subclasses of **`Exception`**
- An exception can be caught by any type higher up in the hierarchy

Exception

StandardError

ArithmeticError

ZeroDivisionError

SystemExit

ValueError

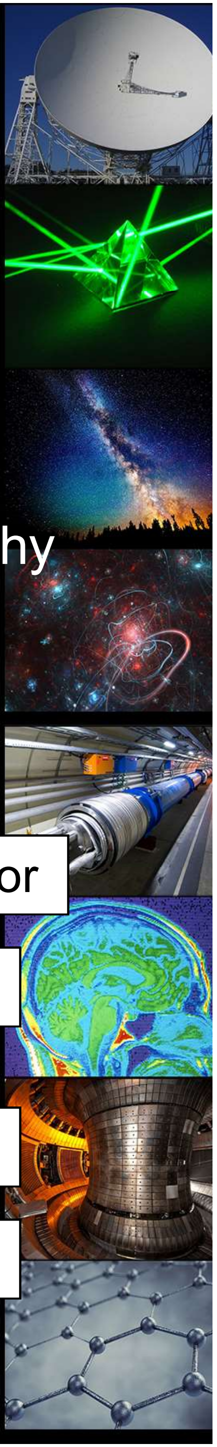
OverflowError

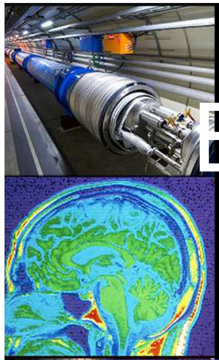
StopIteration

LookupError

IndexError

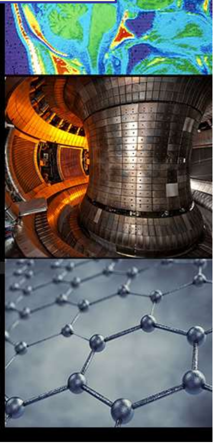
KeyError

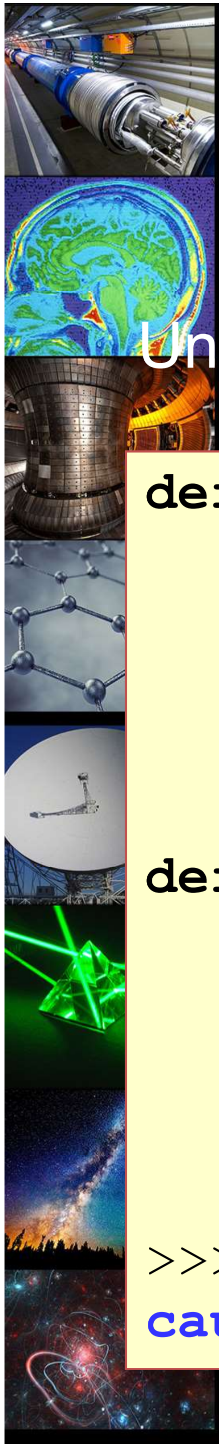




Example of Hierarchical Exception

```
try:
     x = 1 / 0
except ArithmeticError:
     print 'ArithmeticError caught'


ArithmeticException caught
```

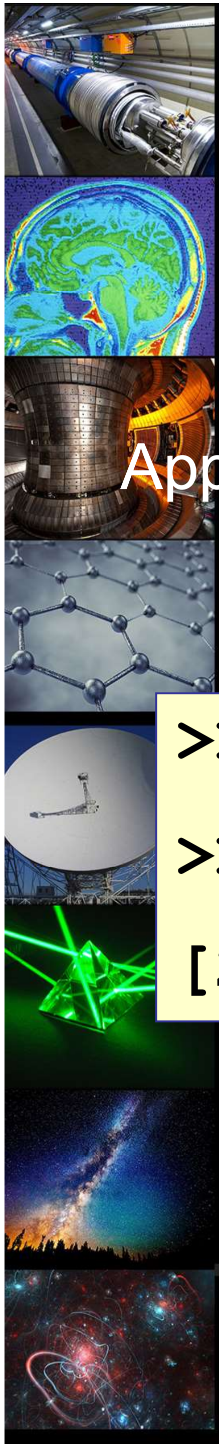




Propagation of Exceptions

Uncaught exceptions propagate up to the calling function

```
def func1():
        try:
             a = 1 / 0
        except ValueError:
             print 'caught by func1'

def func2():
        try:
             func1()
        except:
             print 'caught by func2'

>>> func2()
caught by func2
```



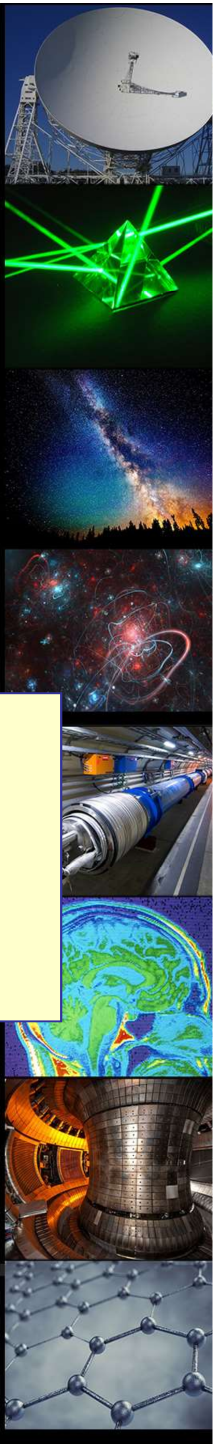
List Comprehensions

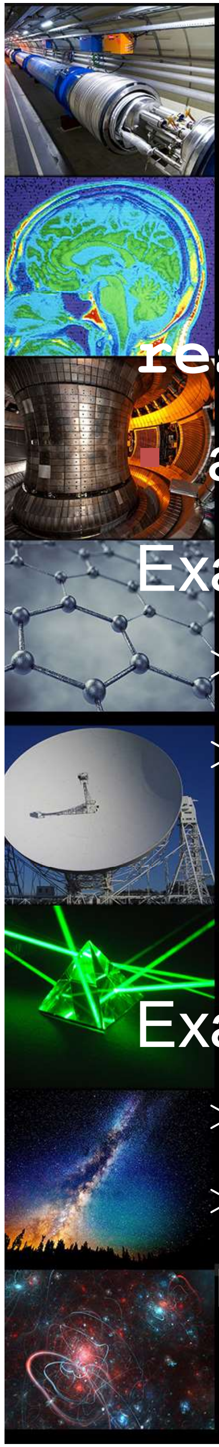
```
expression for var in list ]
```

Apply an expression to every element of a list

- Can simultaneously **`map`** and **`filter`**

```
>>> import math
>>> [math.pow(2,x) for x in range(1,7)]
[2.0, 4.0, 8.0, 16.0, 32.0, 64.0]
```





List Comprehension with filter

result = [*expr* for *var* in *list* if *expr2*]

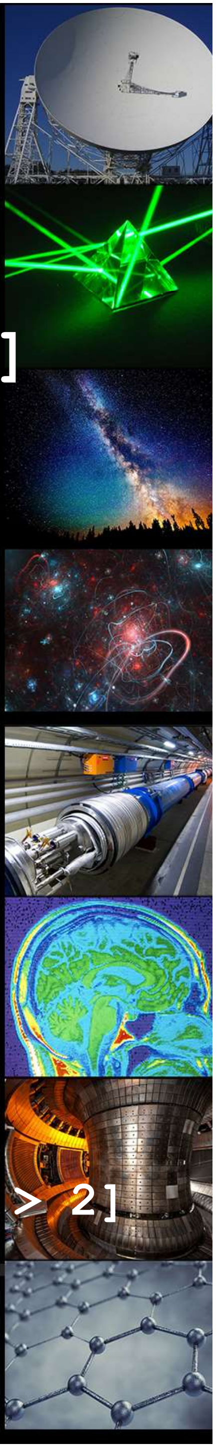
apply to elements of **list** where **expr2** is true

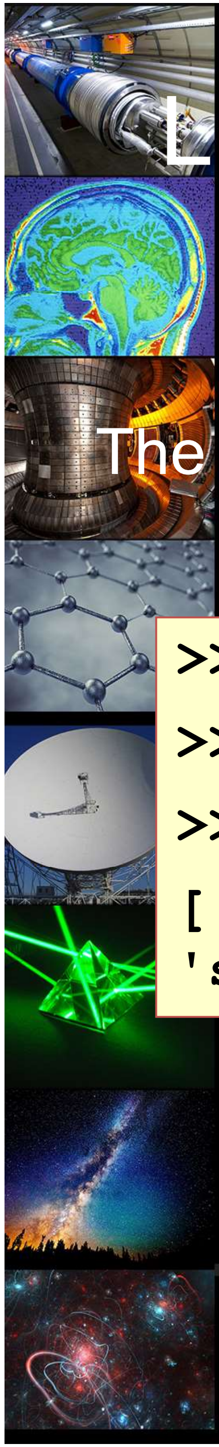
Example: Remove smallest element from list

```
>>> lst1 = [5, 10, 3, 9]
>>> [x for x in lst1 if x != min(lst1)]
[5, 10, 9]
```

Example: Sum all lists of size greater than 2

```
>>> lst1 = [[1, 2, 4], [3, 1], [5, 9, 10, 11]]
>>> [reduce(operator.add, x) for x in lst1 if len(x) > 2]
[7, 35]
```



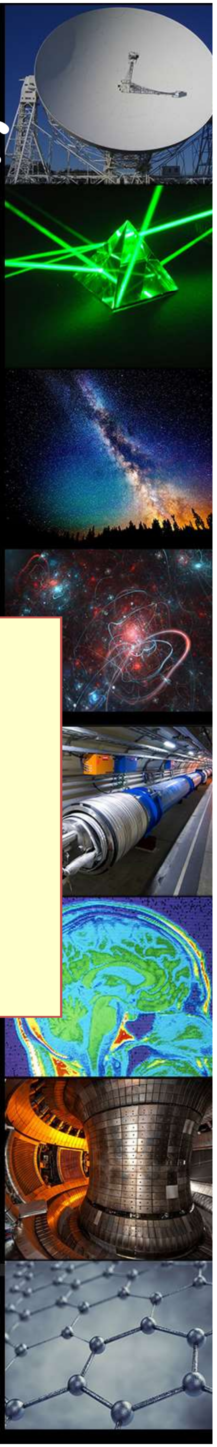


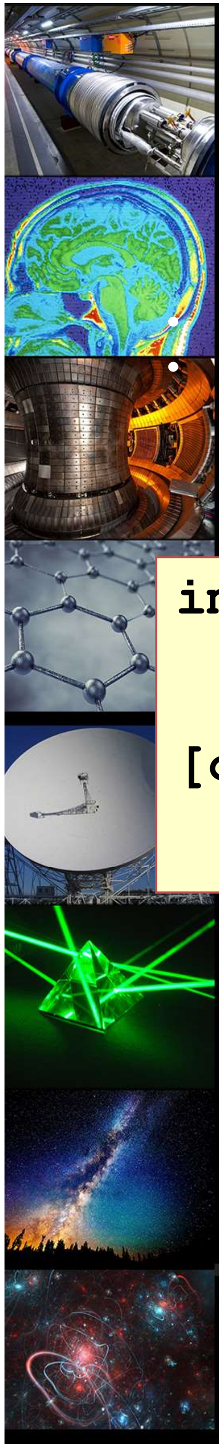
List Comprehension with nested `for`

```
[expr for x in list1 for y in list2]
```

The loops will be nested

```
>>> vowels = ['a','e','i','o','u']
>>> const = ['b','s']
>>> [c+v for c in const for v in vowels]
['ba', 'be', 'bi', 'bo', 'bu', 'sa', 'se', 'si',
'so', 'su']
```

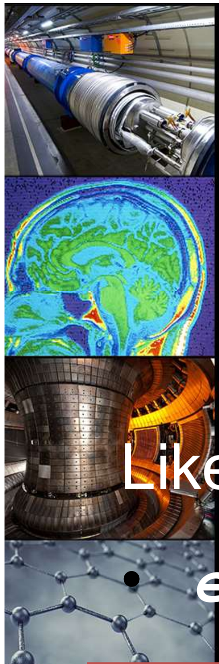




List Comprehension for files

- Find all files in directory larger than 1MB
- Return the real path to file

```
import os, glob


[os.path.realpath(f) for f in glob.glob('*.*')
   if os.stat(f).st_size >= 1000000]
```



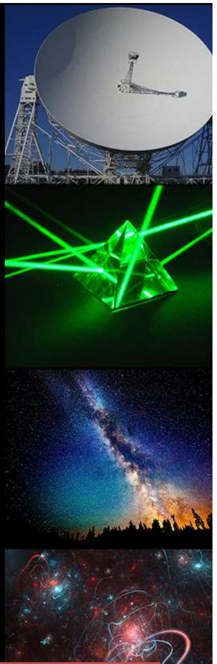
Dictionary Comprehensions

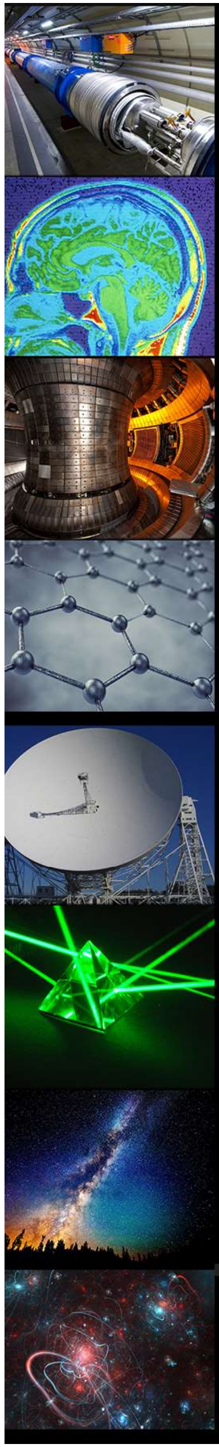
`dict = {`*`expr`*` for var in list if expr2}`

Like list comprehension but generates a dictionary.

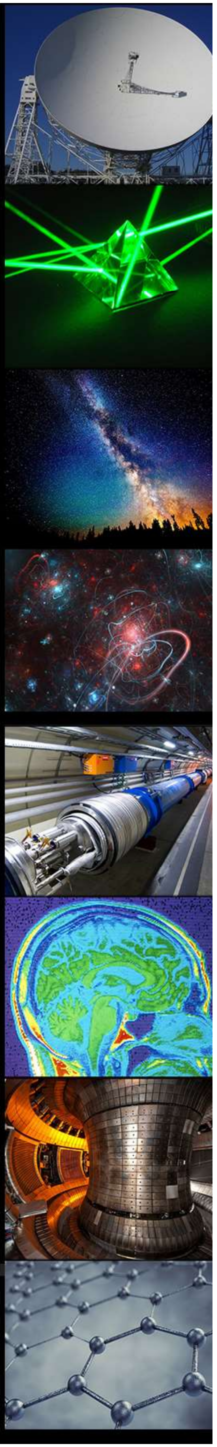
- *expr* must be key:value pair (of course)

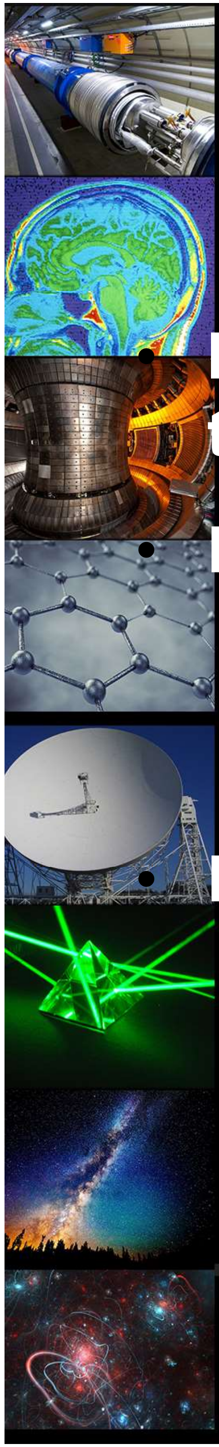
```
# Create dictionary of .exe filenames and sizes.
import os, glob
os.chdir('/windows/system32')
files = {fname:os.stat(fname).st_size
         for fname in glob.glob('*.exe') }

# print them
for (key,val) in files.items():
    print("%-20s  %8d" % (key,val))
```





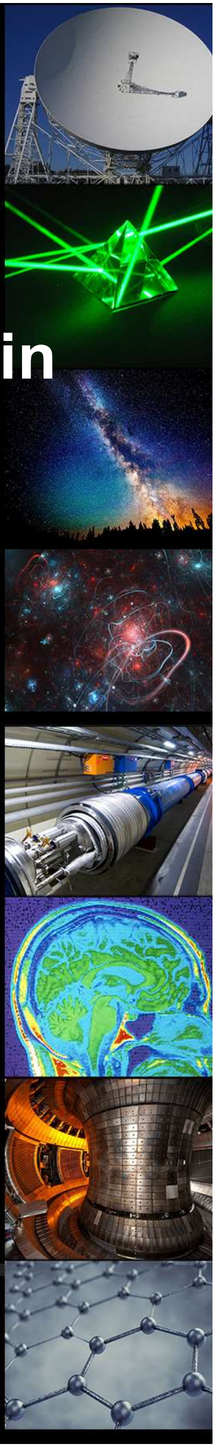
Functional Programming



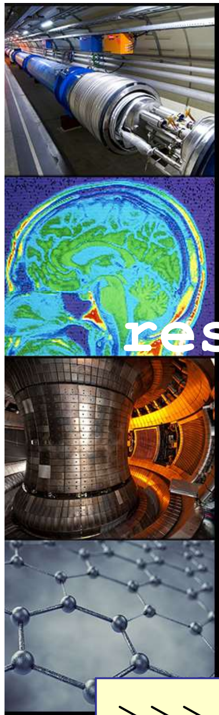


Functional Approaches

- **In Function Programming, functions can be used in the same way as other data types.**
- **Python borrows from functional languages:**
  - Lisp/Scheme
  - Haskell
- **Python built-in functions for functional style:**
  - `map()`
  - `filter()`
  - `reduce()`
  - `zip()`



`map`: "Apply to all" operation



`result = map(func, list)`

func is applied to each element of the list

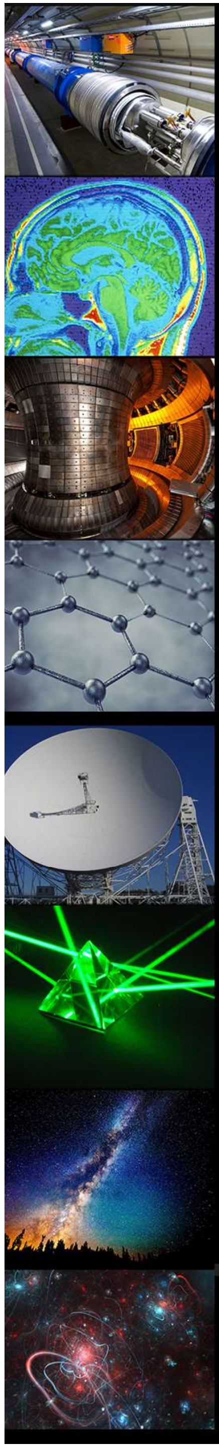
result is a new list, same length as original list

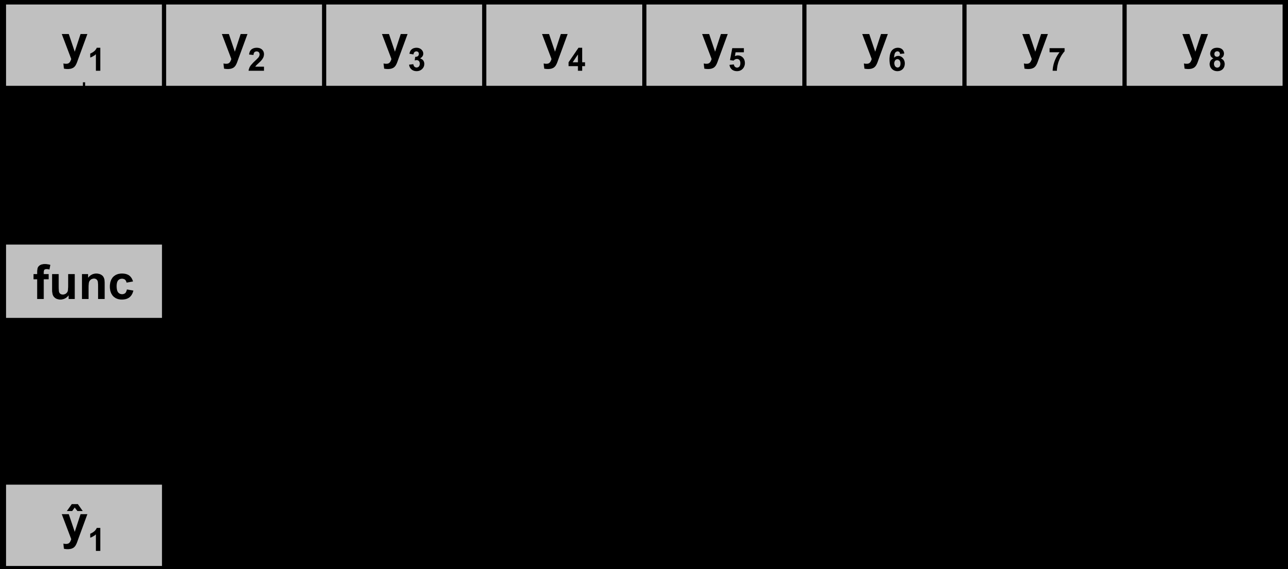
the original list is not altered

```
>>> list = [2, 3, 4, 5]
>>> roots = map( math.sqrt, list )
>>> for x in roots:
       print(x)
1.41421356237
1.73205080757
2.0
2.2360679775
```

Function as argument.

`map` in Action

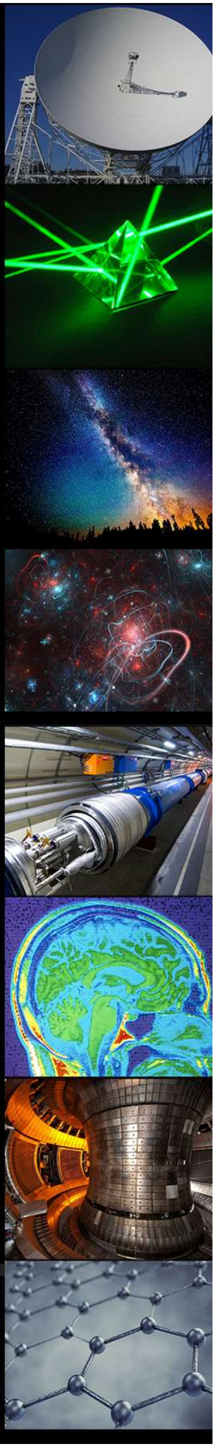




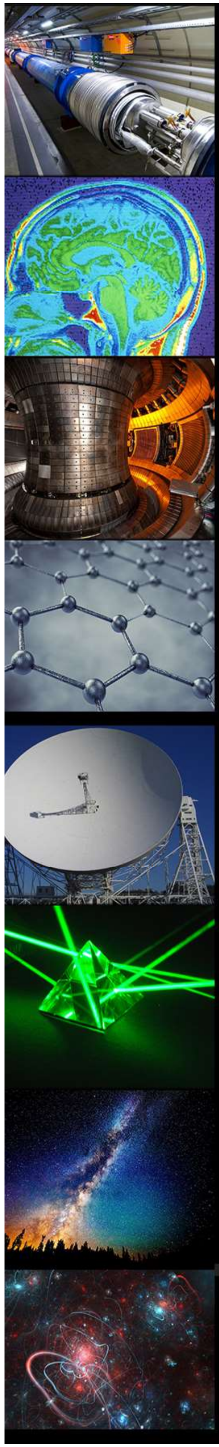
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

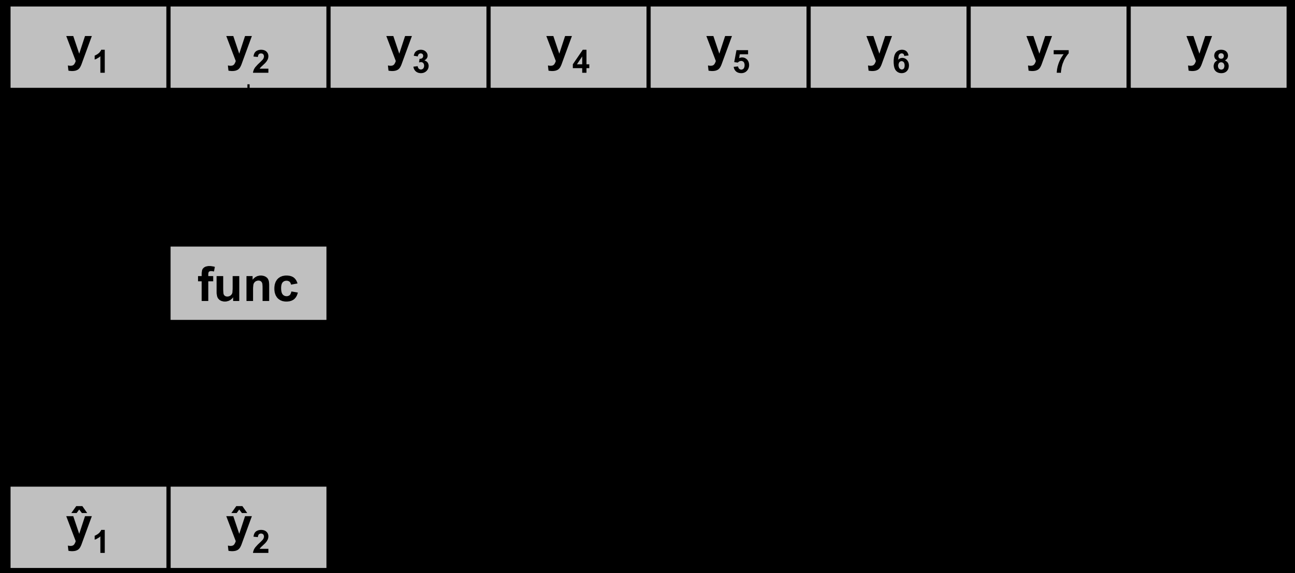
func

$\hat{y}_1$



`map` in Action

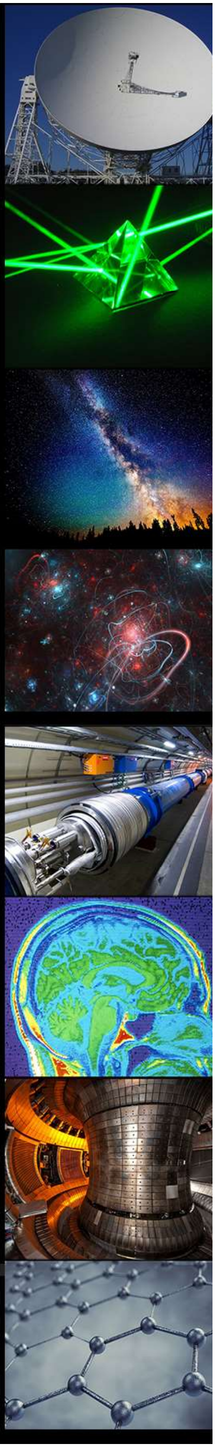




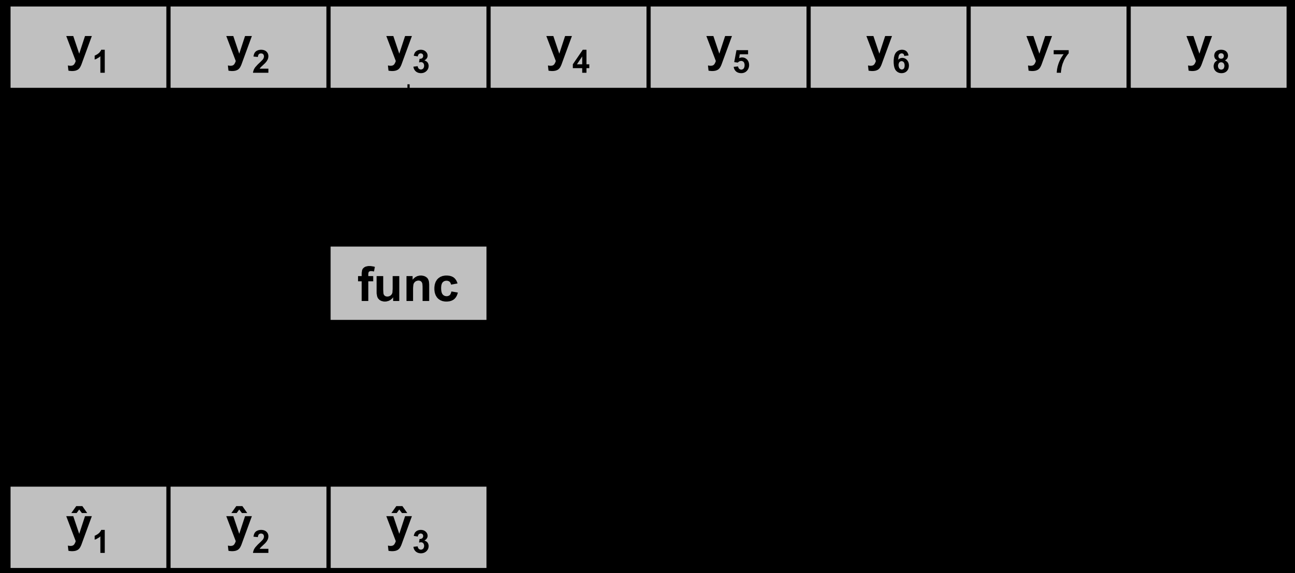
y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8

func

$\hat{y}_1$	$\hat{y}_2$



`map` in Action

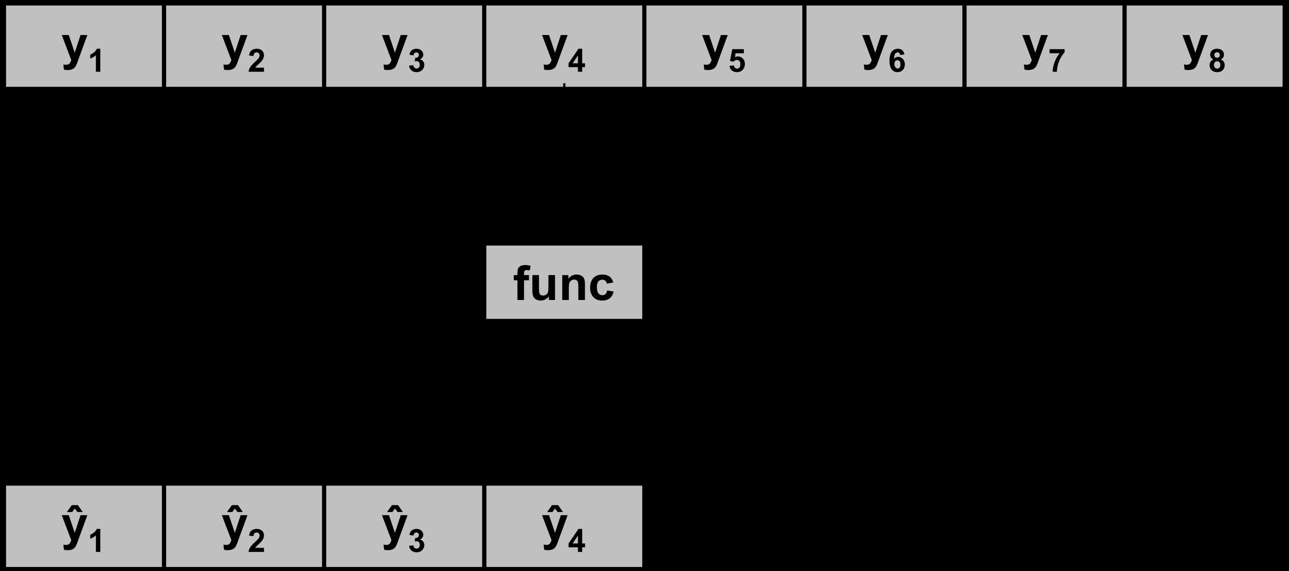


y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

func

$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$

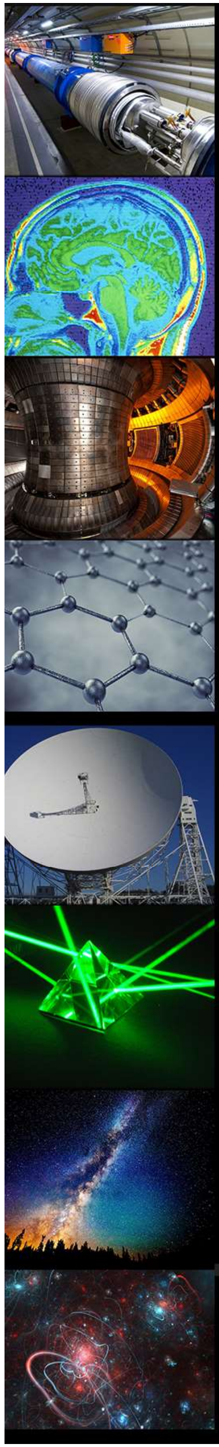
`map` in Action



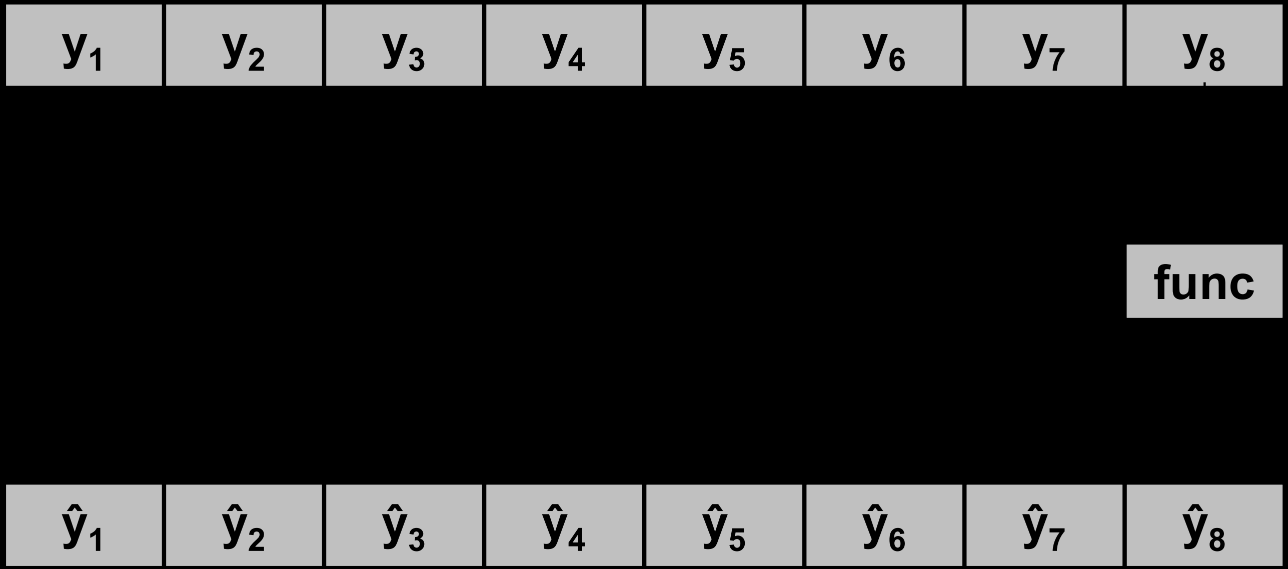
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

func

$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$



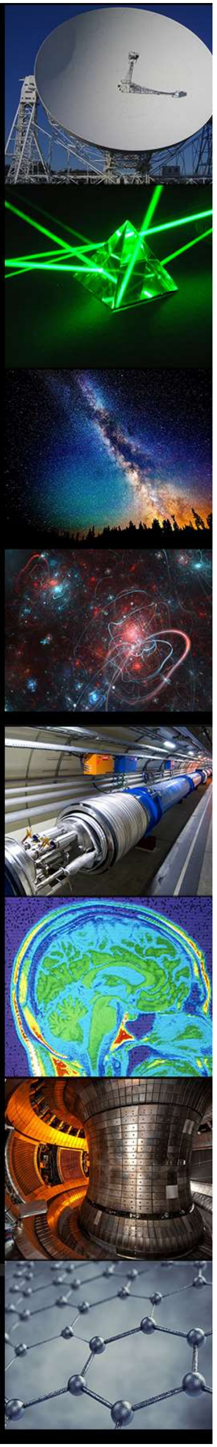
`map` in Action



y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

func

$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$ $\hat{y}_5$ $\hat{y}_6$ $\hat{y}_7$ $\hat{y}_8$



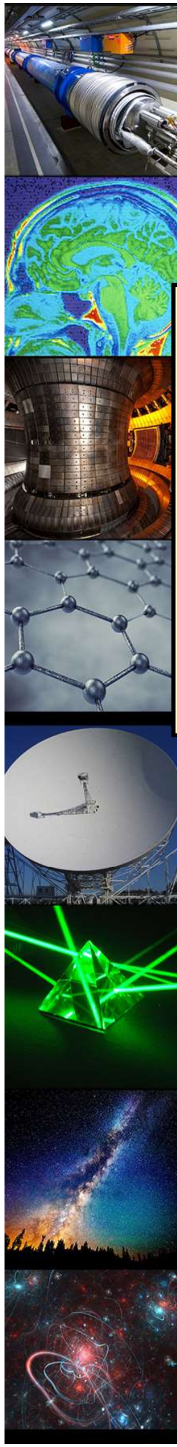
`map` with multiple lists

What if the function requires more than one argument?

$$\texttt{result = map(func, list}_1\texttt{, ..., list}_n\texttt{)}$$

- All lists must be of same length (*not really*)
- Number of lists (*n*) must match #args needed by **`func`**

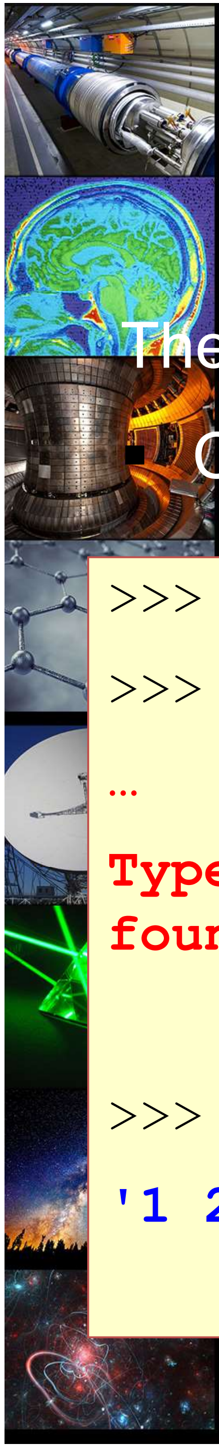
```
# powers of 2
pows = map( math.pow, [2]*5, [1,2,3,4,5] )
# same thing, using a range
pows = map( math.pow, [2]*5, range(1,6) )
```



Use map to reduce coding

```
lst1 = [0, 1, 2, 3]
lst2 = [4, 5, 6, 7]
lst3 = []
for k in range(len(lst1)):
    lst3.append( add2(lst1[k],lst2[k]) )
```

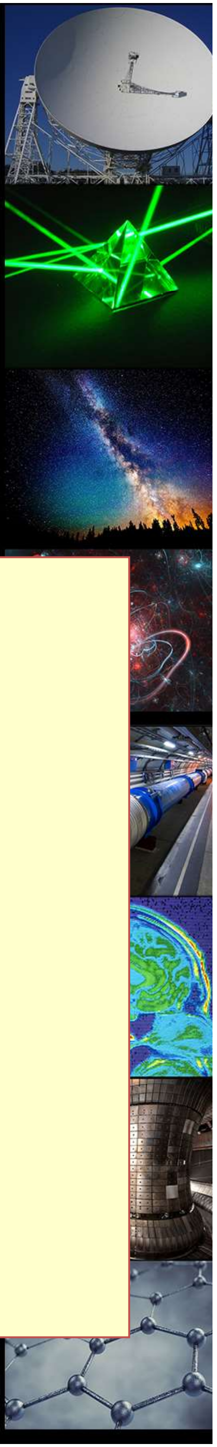
```
lst1 = [0, 1, 2, 3]
lst2 = [4, 5, 6, 7]
lst3 = map(add2, lst1, lst2)
```

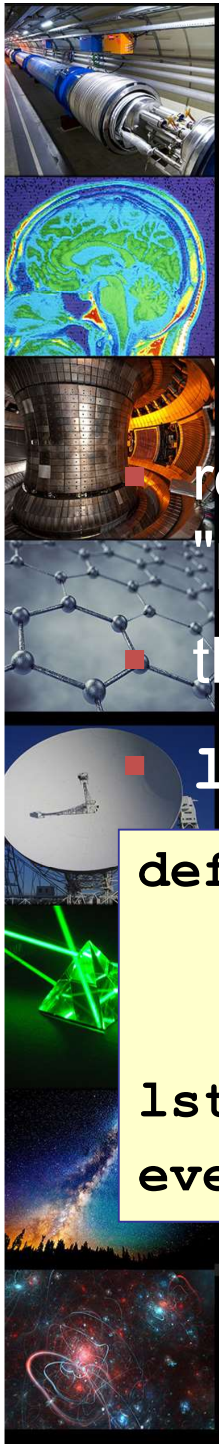


Benefits

- The **map** function can be used like an expression
  - Can be used as a parameter to a function

```
>>> lst1 = [1, 2, 3, 4]
>>> string.join(lst1) # Error: lst1 contains ints
…
TypeError: sequence item 0: expected string, int
found


>>> string.join( map(str, lst1) )  # Correct
'1 2 3 4'
```



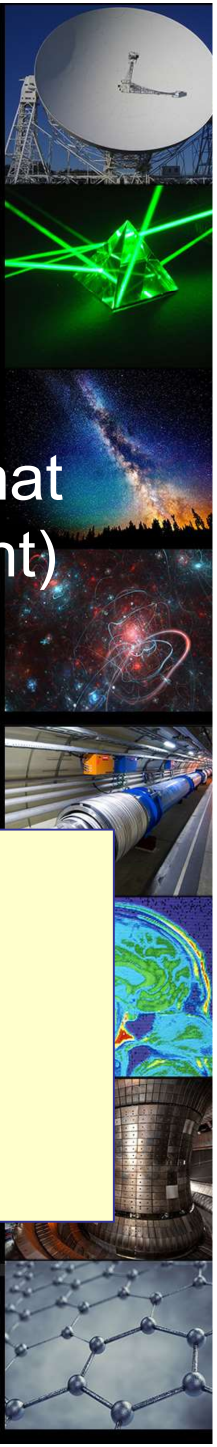


`filter` elements of a list

```
sublist = filter(func, list)
```

- return a sublist from `list` containing only elements that "pass" `func` (func returns True or non-zero for element)
- the result has length less than or equal to the **`list`**
- **`list`** is not altered

```
def isEven(x):
      return x%2 == 0

lst = [2, 7, 9, 8, -12, 11]
even = filter(isEven, lst)  # even = [2, 8, -12]
```

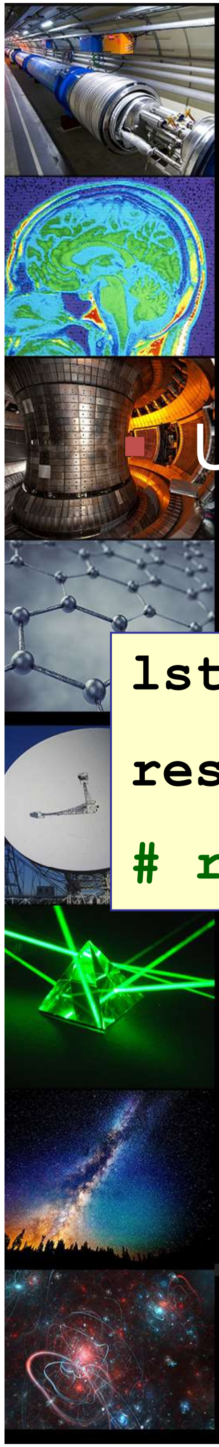


`reduce` accumulate a result

```
result = reduce(func, list)
```

- Apply **`func`** cumulatively to a sequence
- **`func`** must take 2 parameters
- For each element in list, previous **`func`** result is used as 1st arg.

```
def multiply(x, y):
      return x*y


lst = [1,2,3,4,5]
result = reduce(multiply, lst) # result = 1*2*3*4*5
```

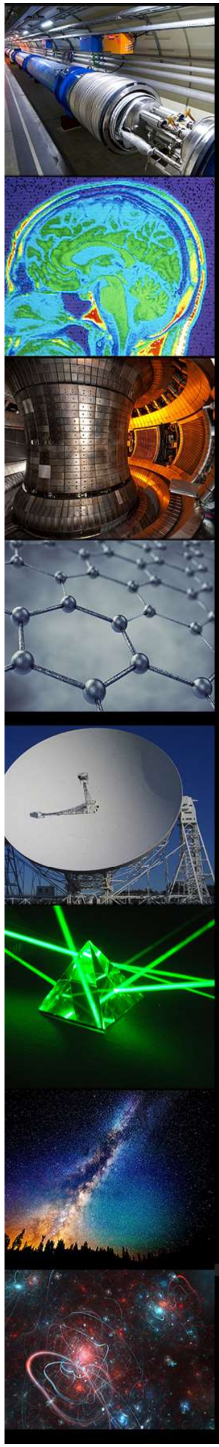


reduce with initial value

reduce(func, initial_value, list)

- Use **initial_value** as parameter in first call to func

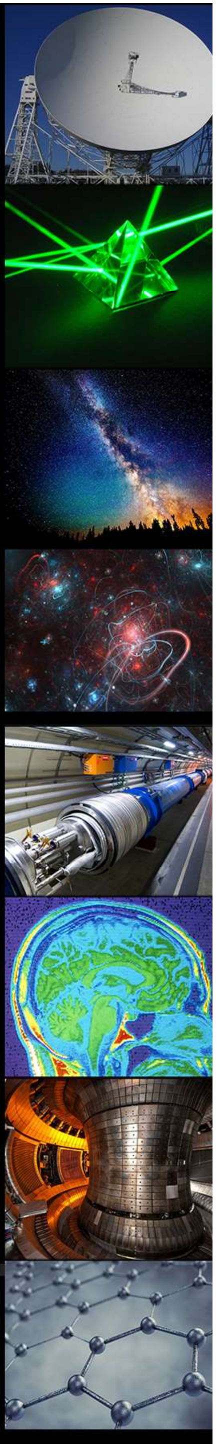
```
lst = [2,3,4,5]

result = reduce(multiply, 10, lst)

# result = 10 * 2 * 3 * 4 * 5
```

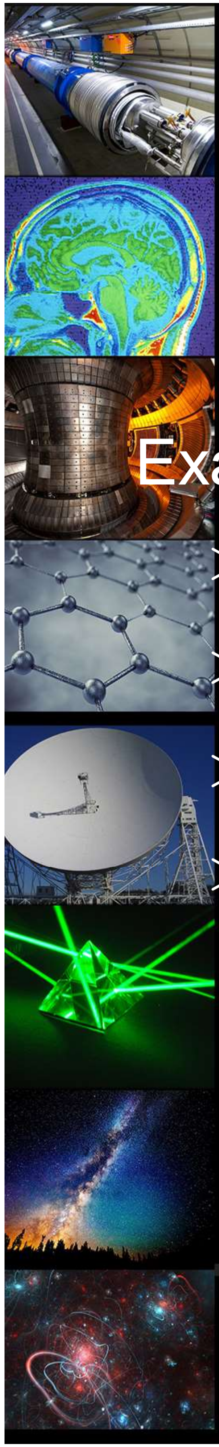


Code Comparison

```
lst1 = [ [2, 4], [5, 9], [1, 7] ]
result = operator.add([100], lst1[0])
for element in lst1[1:]:
result = operator.add(sum, element)
```

```
lst1 = [ [2, 4], [5, 9], [1, 7] ]
result = reduce(operator.add, lst1, [100])
```





Join lists element-by-element

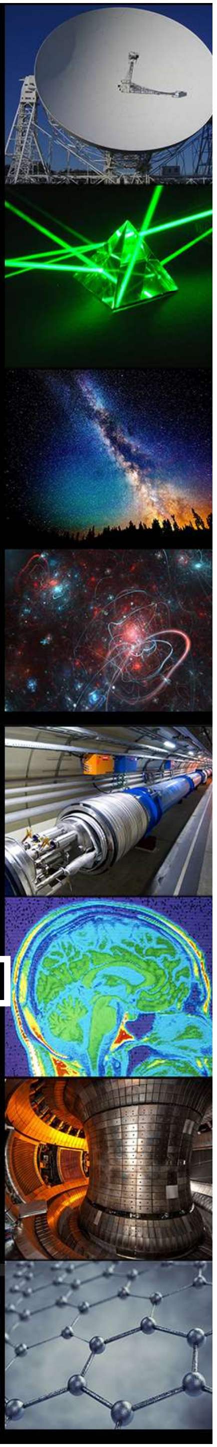
```
zip(list1, …, listn)
```

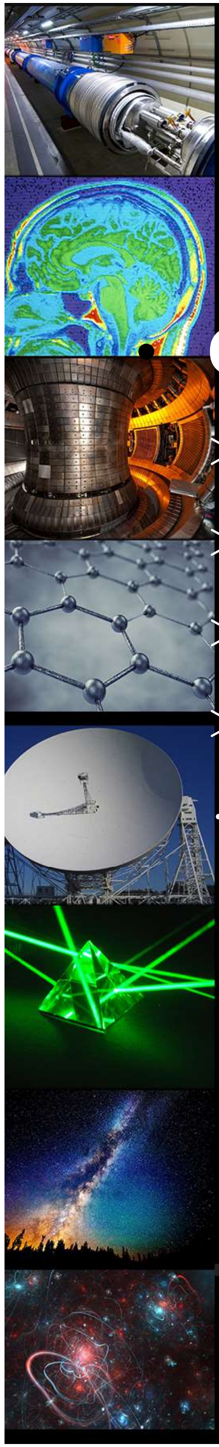
Example: Combine two lists

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = ['a', 'b', 'c', 'd', 'e']
>>> result = zip(lst1, lst2)
>>> result
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

The ‘e’ element was truncated since `lst1` only has 4 elements

The result is a list of tuples

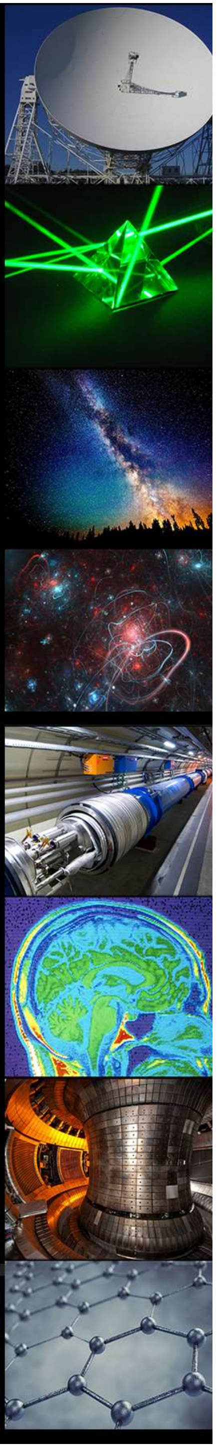




Uses for `zip`

- **Create a dictionary using `zip()` and `dict()`**

```
>>> produce = ['apples', 'oranges', 'pears']
>>> prices = [0.50, 0.45, 0.55]
>>> priceDict = dict(zip(produce, prices))
>>> print priceDict
{'pears': 0.55, 'apples': 0.5, 'oranges': 0.45}
```



Lambda Functions

Anonymous functions. Can be used assigned to variable.

Syntax : **lambda p_1[,…,p_n]:** ***expression***

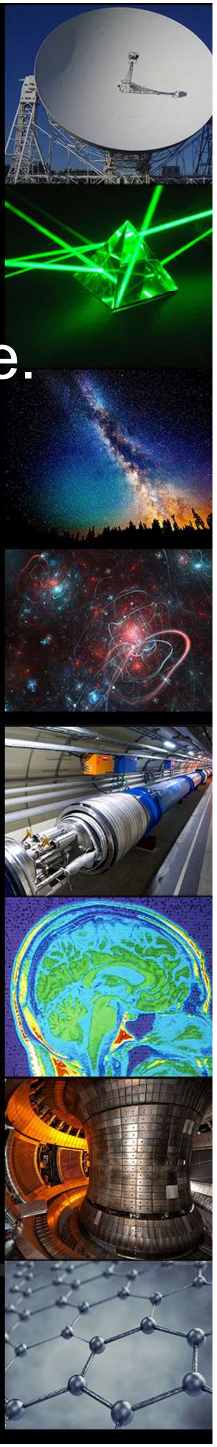
- *expression* should not use **return**

Example: create a square function

```
>>> sqr = lambda x: x*x
>>> print sqr(5)
25
```

Example: average of 2 values

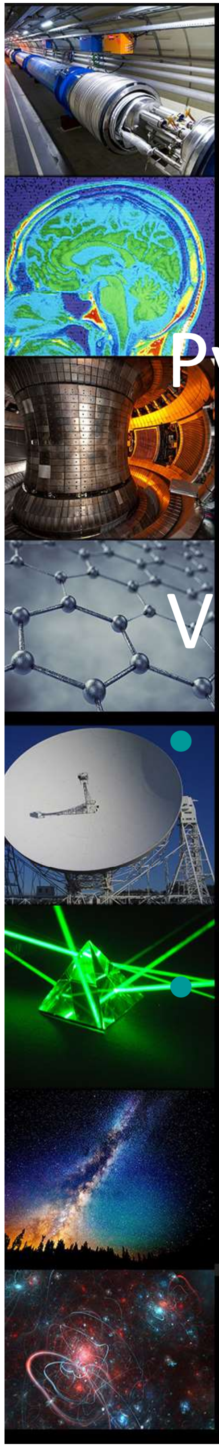
```
>>> avg = lambda x,y: (x+y)/2
>>> avg(2,7)
4.5
```



Function can return a Lambda

Lan

```
# define a power function: only x is "bound" to the lambda
define power(n):
     return lambda x: math.pow(x, n)
cube = power(3)
cube(10)
1000.0
list( map( cube, range(1,10) ) )
[1.0, 8.0, 27.0, ..., 729.0]
```



References

Python Documentation

http://docs.python.org

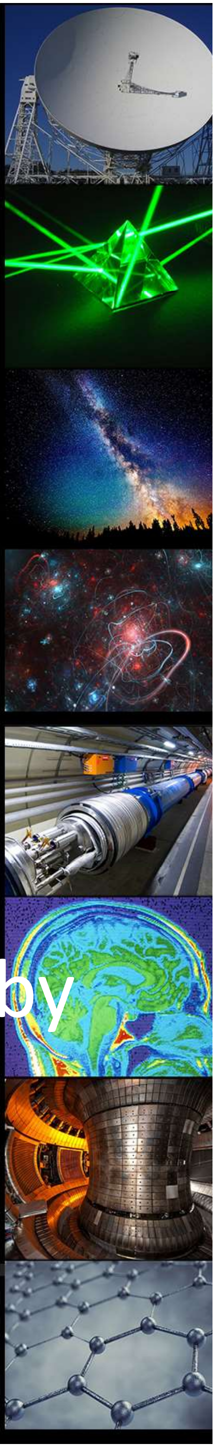
Videos

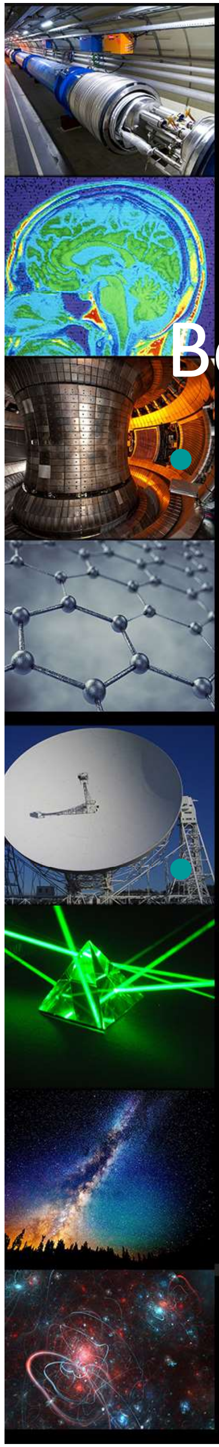
- *Python for Programmers* by Alex Martelli

  http://video.google.com/videoplay?docid=1135114630744003385

- *Advanced Python (Understanding Python)* by Thomas Wouters

  http://video.google.com/videoplay?docid=7760178035196894549





References

Books

- *Practical Programming: an Introduction ...* by Jennifer Campbell, et al. (Pragmatic Programmers, 2009)

  Good book for novice programmer.

- *Python Essential Reference,* 4E. (Addison Wesley)

  Recommended by CPE/SKE students. Has tutorial and language description. More than just a reference.