

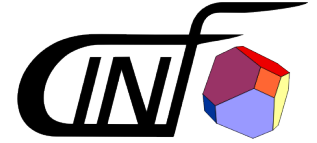


Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

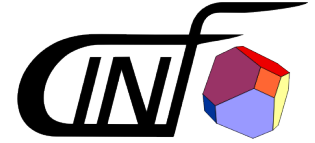
# Python part1

# Outline



- Why Python and what is it?
- The Basics (as they look in Python)
  - Assignments
  - Data types
  - Comparisons
  - Flow control
  - Loops
  - Functions
  - Imports
  - Exceptions

# About Python

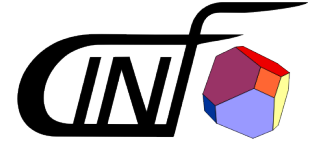


“Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C.”

- Source Wikipedia
- Created by Guido van Rossum in 1991



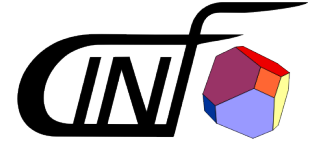
# About Python



“Python is a **widely used** general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C.”

- Source Wikipedia

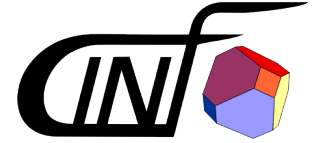
# About Python



“Python is a widely used **general-purpose**, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C.”

- Source Wikipedia

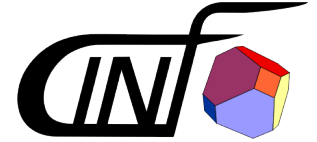
# About Python



“Python is a widely used general-purpose, **high-level** programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C.”

- Source Wikipedia

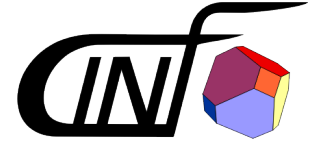
# About Python



“Python is a widely used general-purpose, high-level programming language. Its **design philosophy emphasizes code readability**, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C.”

- Source Wikipedia

# About Python

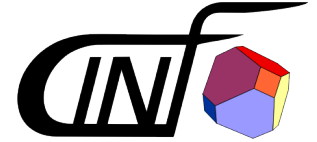


“Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to **express concepts in fewer lines of code** than would be possible in languages such as C.”

- Source Wikipedia

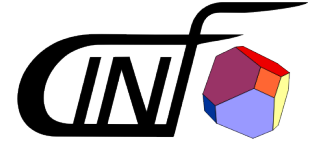


# Why Python?



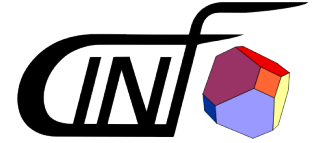
- Easy to learn
- Expressive, easy syntax
  - Indentation is part of the syntax!
- Dynamic typing
- Automatic memory handling
- Interpreted
- Large vibrant community
- Used extensively in scientific community
- Heavy use of great programming concepts
  - Generators, protocols via special methods, etc.
- Python has Zen!

# Why Python?



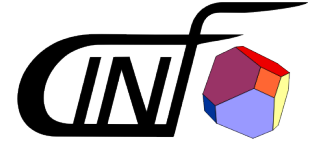
- Easy to learn
- Expressive, easy syntax
  - Indentation is part of the syntax!
- Dynamic typing
- Automatic memory handling
- Interpreted
- Large vibrant community
- Heavy use of great programming concepts
  - Generators, protocols via special methods, etc.
- Easy to write
  - Fast prototyping
  - Good for one-off scripts
- Focus on readability
- Used extensively in scientific community
  - Many packages available
- Python has Zen!

## Exercise time



- Open your interactive terminal

## Exercise time

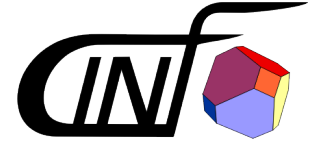


- Open your interactive terminal
- Execute the command:  

```
>>> import this
```

# The Zen of Python

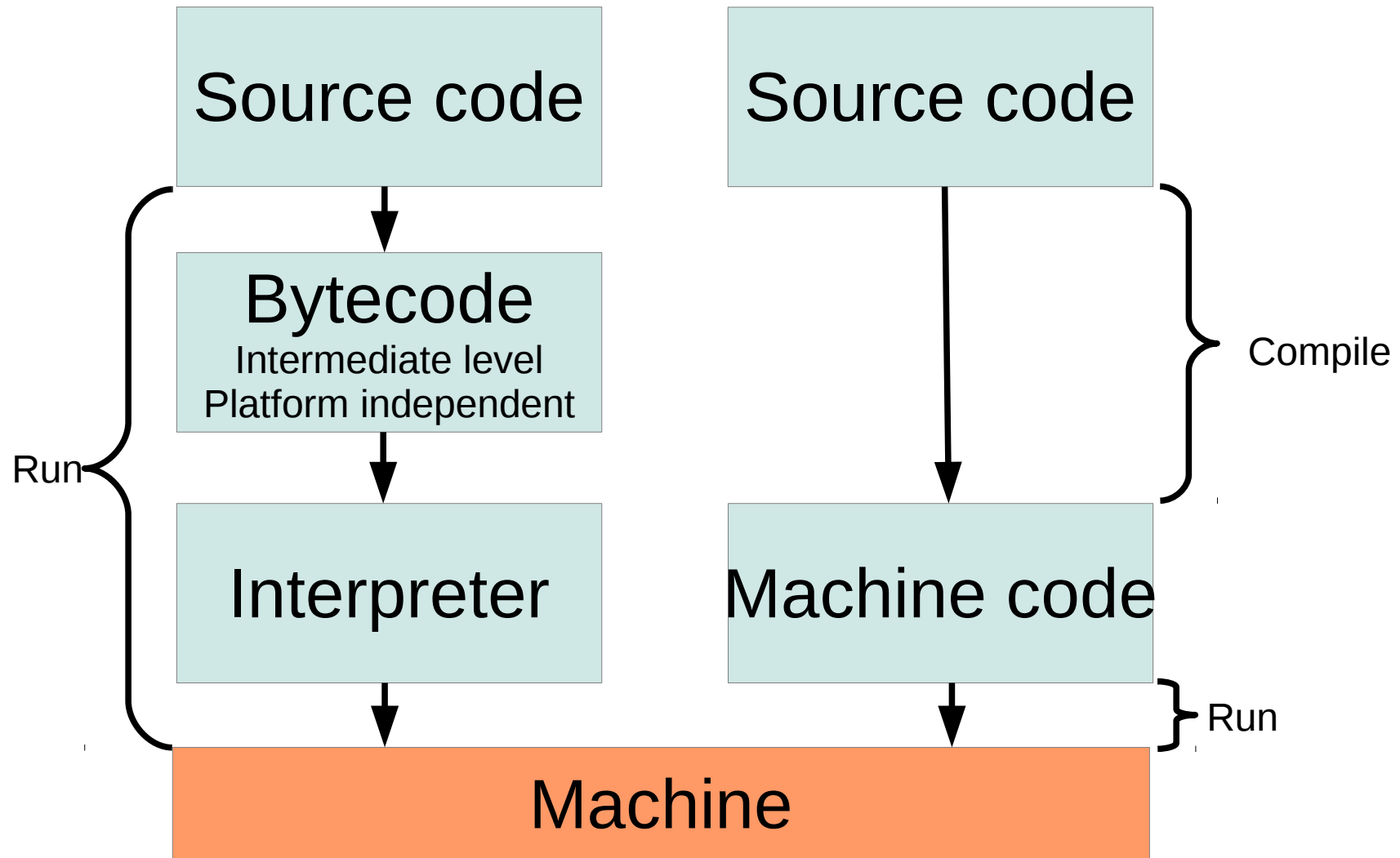
we will return to this



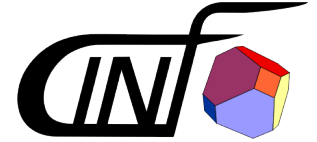
```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

# It is interpreted



# Indentation is part of the syntax



- At first it seems weird
- Then you love it
- You indent anyway!
- Reduces syntax verbosity
- Eliminates erroneous reading due to bad indentation

```
for n in range(10):  
    print('Hallo World!')
```

```
>>> colors = ['magenta', 'cyan', 'crimson']
>>> for color in colors:
...     print(color)
...
magenta
cyan
crimson
```

The block  
of code

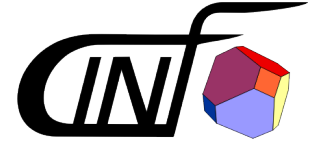
Notice the : that indicates that a  
block of code is about to start

The spaces indicate a block



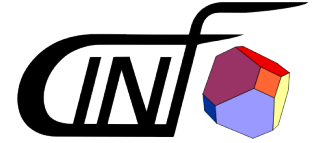
Enough sales talk  
Let's get started

# The structure

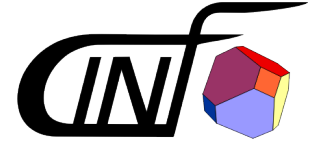


- Move through each of the topics in the outline
- For each topic there will be:
  - A presentation by me
  - A “type along” session, where we type the new things together
  - A few exerciser (the exercises are meant mainly for typing repetition, so most are simple)
- There will also be a few short detours along the way
- A larger exercise towards the end

# Just a kick starter ...



## About depth in the topics



- I love talking about programming
- There are many advanced topics that I would love to share
- Not feasible (nor practical) for this format and scope
- Will mark advanced topics as “Ask me over coffee” topics





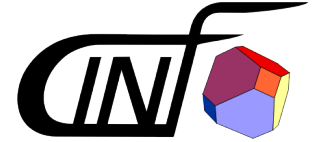
Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1

# Data types

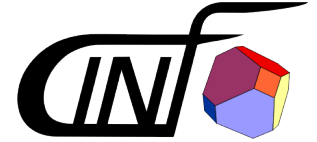
the usual suspects



Type	Type functions	Example value
Integer	int()	1 or 42
Float	float()	1.0 or 42.0
String	str()	'Hey!'
Boolean	bool()	True or False
Complex number	complex()	(1+42j)
No value	---	None

Plus some more we will not worry about now; e.g. fractions and decimal types

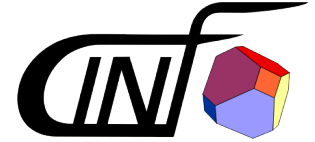
# Data types



```
>>> int(1)
1
>>> float(1)
1.0
>>> str(1)
'1'
>>> bool(1)
True
>>> complex(1)
(1+0j)
```

Code examples from <https://docs.python.org/3/tutorial/>

# Expressions and assignments



Action	Operator
Add	+
Subtract	-
Multiply	*
Division (careful)	/
Explicit floor division	//
Raise to power	**
Assignment	=

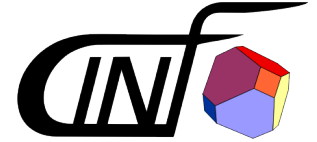
The output type generally is the most flexible type in the expression

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> 17 / 3 # int / int -> float
5.666666666666667
>>> 17 // 3 # int // int -> int
5
>>> 17 / 3.0 # int / float -> float
5.666666666666667
>>> 5 ** 2 # 5 squared
25
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Code examples from  
<https://docs.python.org/3/tutorial/>

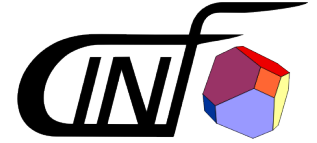


# Variables



- Variables are nothing more or less than name tags
  - `my_variable = 20`
- Means, take the integer with the value of 20 and give it the name “my\_variable”
  - `soda_in_case = 4 * 6 * 0.33`
- Means calculate  $4 * 6 * 0.33$  and give the resulting float the name “soda\_in\_case”
- Objects can have more than one name

# Conditions

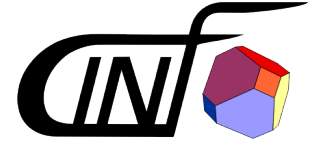


- Expressions that evaluate to True or False
  - E.g:  $4.0 < 5.5$  or  $8 == 8$

Operator	Explanation
<	less than
>	greater than
<=	less than or equal
>=	more than or equal
!=	not equal (careful with floats)
==	is equal
is	is the same object
in	is contained in

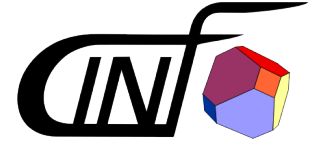


# Conditions



- “not” is used to negate
- Can always be applied before an expression
  - `not 4 < 10`
- Can also be applied before “in” and after “is” (for readability)
  - `4 not in [1, 2, 3 4]`
  - `not 4 in [1, 2, 3, 4]` # Is also valid
  - `not True is False`
  - `True is not False` # also valid

# Boolean operators



- Boolean values and expressions can be chained together with “and” and “or”
  - **and** True of both left and right side is True
  - **or** True if either side is True
- Use **any()** and **all()**, for compound statements

```
0 < n and n % 2 == 0
```

```
n == 42 or n == 47 or n == 0
```

```
all([True, True, False])
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- Try a few assignments:
  - `variable1 = 5`
  - `variable2 = 1.5`
  - `variable3 = 'Hi There'`
- Check their type:
  - `print(type(variable1))`
- Try a bit of math, both with variables and just numbers
  - `39 + 3`, `variable1 + variable2`,  
`variable1 * variable2`
- And looks what happens with:
  - `variable1 * variable3`
- Make a boolean check for whether a number is in a float range



Center for Individual Nanoparticle Functionality

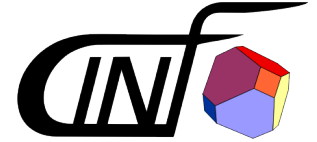
- Variables, types, comparisons
- **Strings**
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1

- (Text) strings are at the center of many tasks
- Even in scientific computing
  - (metadata, text based export data file)
- Titles, labels and legends for plots
- Status prints during data processing



# Strings

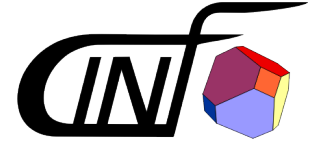


```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
```

```
>>> s = 'First line.\nSecond line.'
>>> s
'First line.\nSecond line.'
>>> print(s)
First line.
Second line.
```

Code examples from <https://docs.python.org/3/tutorial/>

# Strings, multi-line and concatenation



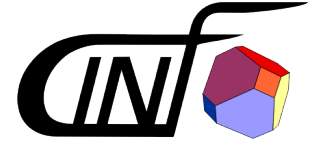
```
>>> print("""\nMultiline strings\nare strings that ...\nspan multiple lines\n""")\nMultiline strings\nare strings that ...\nspan multiple lines
```

```
>>> 3 * 'un' + 'ium'    # Multiplication behavior important\n'ununinium'
```

```
>>> text = ('Put several strings within parentheses '\n            'to have them joined together.')\n>>> text\n'Put several strings within parentheses to have them joined together.'
```

Code examples from <https://docs.python.org/3/tutorial/>

# Strings, indexing and slicing

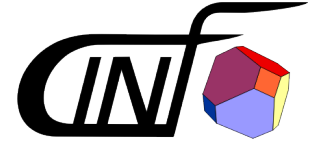


```
>>> word = 'Python'
>>> word[0]
'p'
>>> word[-1]
'n'
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'

>>> len(word)
6
```

Code examples from <https://docs.python.org/3/tutorial/>

# Strings are immutable



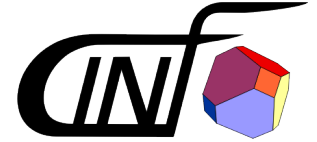
- They cannot be changed in place
- Assignment to an index gives an error

```
>>> string = 'Python'
>>> string[0] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- In-place add creates a new string

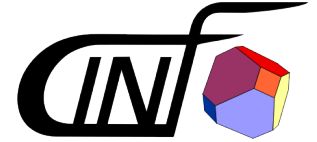
```
>>> string = 'P'
>>> id(string)
140340417585200
>>> string += 'ython'
>>> string
'Python'
>>> id(string)
140340349305024
```

# Strings have lots of methods



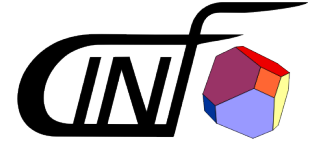
```
>>> string = '...Python...'
>>> dir(string)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
 '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
 '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> string.strip('.')
'Python'
```

## Wait a minute, methods



- Methods are like functions, but they are “attached” to an object, that they “act on”
- `my_string = 'Hello World'`
- `my_string.upper()`
- Means:
  - Create a string object and give it the name `my_string`
  - Call the method “upper” on this string

# Strings have lots of methods



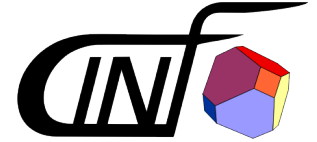
```
>>> string = '...Python...'
>>> dir(string)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
 '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
 '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> string.strip('.')
'Python'
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

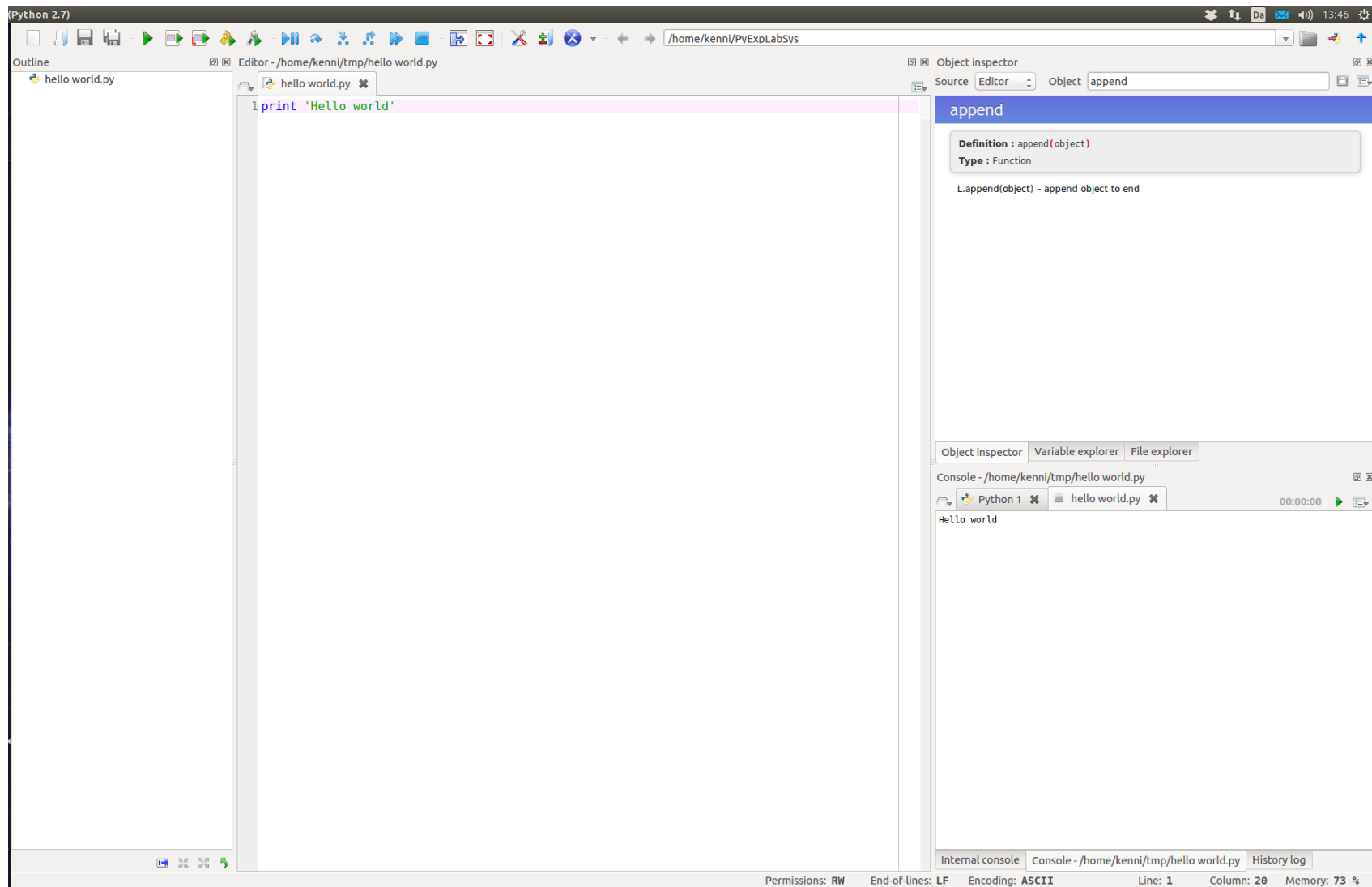
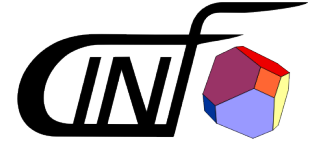


## Execute python code in a file



- Python code can also be executed from a file
- Type the code into a text file and execute it with:
  - `python my_code.py`
- Most integrated development environments IDEs (like e.g. Spyder) has a shortcut for running the current file (often F5)
- This really is the default for most execution of Python code, the terminal is mostly for experimentation
- Now back to the exercise

# Spyder

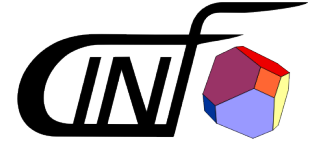


# Exercise time

- Do exercise `extract_from_string.py`
- Try the strip method `'Python ready'.strip('Py')`
  - What happens?

# Strings have lots of methods

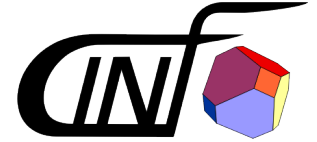
## Use them!



```
>>> statement = 'Python rules'
>>> test = 'Python'
>>> statement.startswith(test)      # is easy to understand
True
>>> statement[:len(test)] == test   # is just horrible
True
```

# The Zen of Python

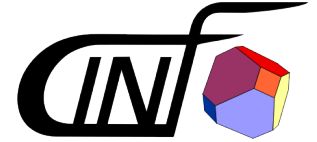
once again



```
>>> import this
The Zen of Python, by Tim Peters
```

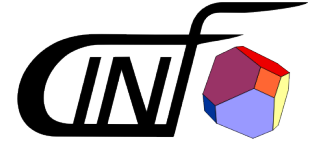
```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

# String formatting



- Used to neatly format other strings and numbers into a string
  - Beyond: center, rjust, ljust
- A common operation for which convenience functions are made
- Makes use of {} format fields in the string and the format method on the string

# String number formatting examples



```
>>> 'An integer: {}'.format(10)
'An integer: 10'
>>> 'Padded with space: {0: >4}'.format(10)
'Padded with space:   10'

>>> 'Format floats: {}'.format(10.0/3)
'Format floats: 3.33333333333'
>>> 'Format floats: {:.2f}'.format(10.0/3)
'Format floats: 3.33'

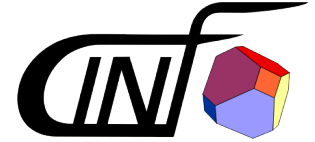
>>> 'To exponential format: {:.3e}'.format(123456 * 10**6)
'To exponential format: 1.235e+11'

>>> 'To percent: {:.2%}'.format(42/47.0)
'To percent: 89.36%'

>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Code examples from <https://docs.python.org/3/tutorial/>

# String formatting examples



```
>>> # Anonymous format fields, used in order
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"

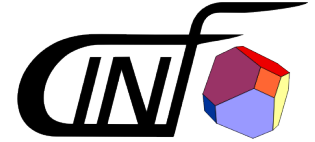
>>> # Numbered anonymous format fields use the argument order
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam

>>> # Named format fields use key-word arguments
>>> print('This {food} is {adjective}.'.format(
...     food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Code examples from <https://docs.python.org/3/tutorial/>



# String number formatting examples



```
>>> 'An integer: {}'.format(10)
'An integer: 10'
>>> 'Padded with space: {0: >4}'.format(10)
'Padded with space:   10'

>>> 'Format floats: {}'.format(10.0/3)
'Format floats: 3.33333333333'
>>> 'Format floats: {:.2f}'.format(10.0/3)
'Format floats: 3.33'

>>> 'To exponential format: {:.3e}'.format(123456 * 10**6)
'To exponential format: 1.235e+11'

>>> 'To percent: {:.2%}'.format(42/47.0)
'To percent: 89.36%'

>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Code examples from <https://docs.python.org/3/tutorial/>

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

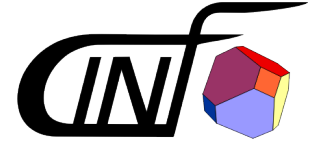
- Format the 10 first number, squares and cubes into a nice looking table:

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Code examples from <https://docs.python.org/3/tutorial/>

- Reverse the order of squares and cubes changing only the string
- <https://docs.python.org/3/library/string.html#formatspec>

# Old string formatting style



- Just forget about it!



```
>>> import math
>>> print('The value of PI is approximately {:.3f}'.format(math.pi))
The value of PI is approximately 3.142.
```

Code examples from <https://docs.python.org/3/tutorial/>

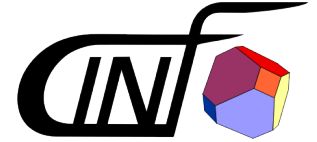


Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- [Lists](#)
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

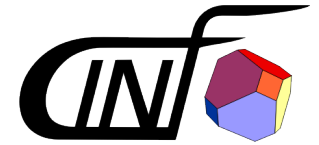
# Python part1

# Lists



- Container objects that are:
- In-homogeneous
  - Can contains a mix of objects of different types  
`mylist = [42, 'is', True, [1,2,3]]`
- Mutable
  - Their content can be modified
- Indexed by numbers
- Fast appending and popping (at one end) and indexing, but unsuitable for accessing large quantities of numeric data in order





# List indexing

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

```
>>> squares[0]
1
>>> squares[-1]
25
>>> squares[-3:]
[9, 16, 25]
```

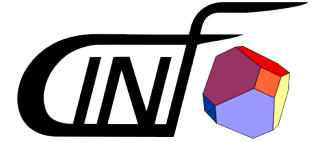
```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3                      # the cube of 4 is 64, not 65!
64
```

```
>>> cubes[3] = 64
>>> cubes
[1, 8, 27, 64, 125]
```

Important: `[]` means, the element for this index

Code examples from <https://docs.python.org/3/tutorial/>

# List additions and alterations



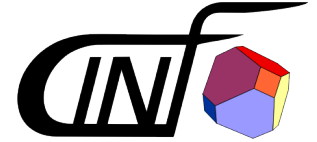
```
>>> squares = [1, 4, 9, 16, 25]
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> squares.append(36)
>>> squares
[1, 4, 9, 16, 25, 36]
>>> squares.pop()
36
```

Code examples from <https://docs.python.org/3/tutorial/>



# Searching lists and their length

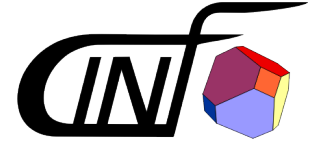


```
>>> mylist = ['Python', 'is' , 'cool', 'and', 'is', 'pretty']
>>> mylist.index('is')
1
>>> mylist.count('is')
2
>>> 'Python' in mylist
True
```

Important, but be careful

```
>>> len(mylist)
6
```

# More on lists



Built-in methods; append, extend, insert, remove, pop, index, count  
Built-in methods in place; sort, reverse

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
```

Code examples from <https://docs.python.org/3/tutorial/>

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- Make a small list
- Get an item out by indexing
- Change an item by indexing
- Try slicing on a list
- Make a list with two elements `mylist=[42, 47]`
  - Try the following: `myvar1, myvar2 = mylist`  
What happens?
- More exercises on lists in the loops section

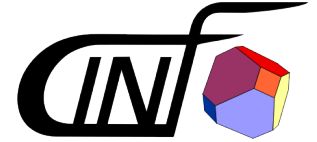


Center for Individual Nanoparticle Functionality

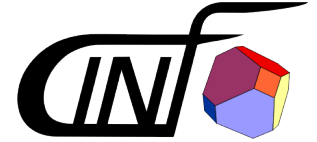
- Variables, types, comparisons
- Strings
- Lists
- **Loops**
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1

# Loops



- Generic programming concept
- Allows you to execute the same block of code several times
- (changing a variable for each execution)
- Commonly a “for” and a “while” loop

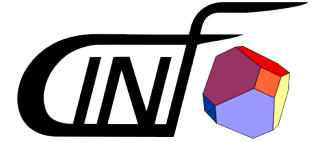


- For and while

```
>>> colors = ['red', 'green', 'blue']
>>> for color in colors:
...     print(color)
...
red
green
blue
```

```
>>> n = 3
>>> while n > -1:
...     print(n)
...     n -= 1
...
3
2
1
0
```

# break

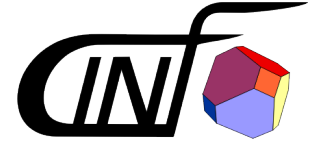


- Break out of the loop  
(without executing the rest of the body)

```
>>> for n in range(10):  
...     if n < 3:  
...         print(n)  
...     else:  
...         break  
...     print('count')  
...  
0  
count  
1  
count  
2  
count
```



# continue

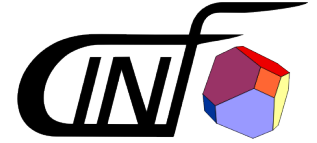


- Continue with the next iteration  
(without executing the rest of the body)

```
>>> for n in range(6):  
...     if n % 2 == 0:  
...         continue  
...     print(n)  
...  
1  
3  
5
```

# A bit about built-in names concerning loops

the blemish on the otherwise Mona Lisa like beautiful face of Python



- 'for' really should have been called 'foreach' to accurately convey what it does
  - But 'for' is shorter so it is not so bad

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- Make a loop over a list of colors and print them inside
- Write an (manual) algorithm that looks for an element in a list
  - Stops and sets the result in a variable if it finds it
  - Sets the variable to a default value if it does not

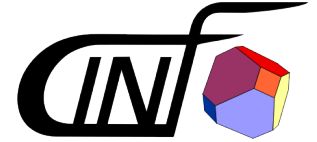


Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- **if-elif-else**
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1

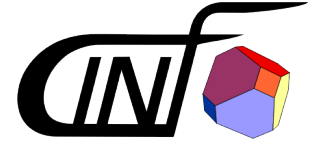
# if-elif-else



- Used to execute different blocks of code depending of a condition
- **elif** is short for “else if”

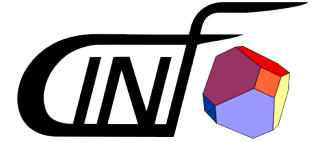
```
if condition:  
    execute this block  
elif this_condition: # elif is short for “else if”  
    then execute this block  
else:  
    if all else fails execute this block
```

# The structure of if-elif-else



- The complete allowed structure is:
  - 1 **if**
  - Followed by 0 or more **elif**
  - Followed by 0 or 1 **else**

# 'if'-'elif'-'else'

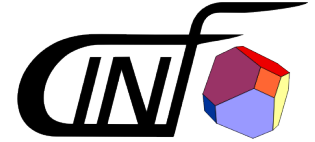



```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Code examples from <https://docs.python.org/3/tutorial/>



# Python has no “switch/case”



- Switch/Case is a programming construct in many other languages
- Just use if-elif-elif-elif-else
- Or use a dict with functions 

```
my_variable = complicated_expression
if my_variable < -1:
    # Do something in this
elif my_variable == 0:
    # Do something in this case
elif my_variable == 1:
    # Do something in this case
else:
    # Do something if nothing
    # else matches
```

# DOES NOT EXIST

```
switch complicated_expression:
    case < -1:
        # Do something in this case
    case 0:
        # Do something in this case
    case 1:
        # Do something in this case
    else:
        # Do something if nothing
        # else matches
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- For the list of words ['Python', 'is', 'awesome']
  - Combine a for loop with an if statement to print every word from the list that contains an 's'.
  - Prints any word in the list that is exactly 7 characters long in all upper case

A few random items

range  
pass

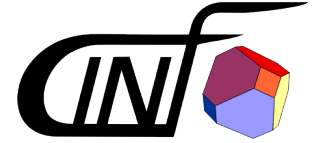
# The 'range' function

- For iterating over integers
- Returns a list of numbers  
`range(from, to_but_not_including, increment)`

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(4, 6))
[4, 5]
>>> list(range(4, 1, -1))
[4, 3, 2]
```

```
>>> for i in range(3):
...     print i
...
0
1
2
```

# The 'pass' statement



- A do-nothing statement
  - For cases where there must be code
- Also used during 'stubbing'

```
for n in range(10):  
    pass    # implement later  
  
def my_function():  
    pass    # fancy function that does absolutely nothing
```

# Exercise time

- Use range and for to loop over every third number going downwards from 10 to 1
- [10, 7, 4, 1]



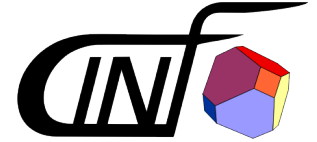
Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- **Functions**
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1

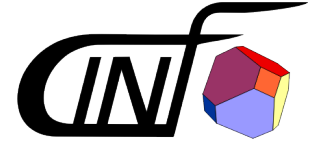


# What is a function

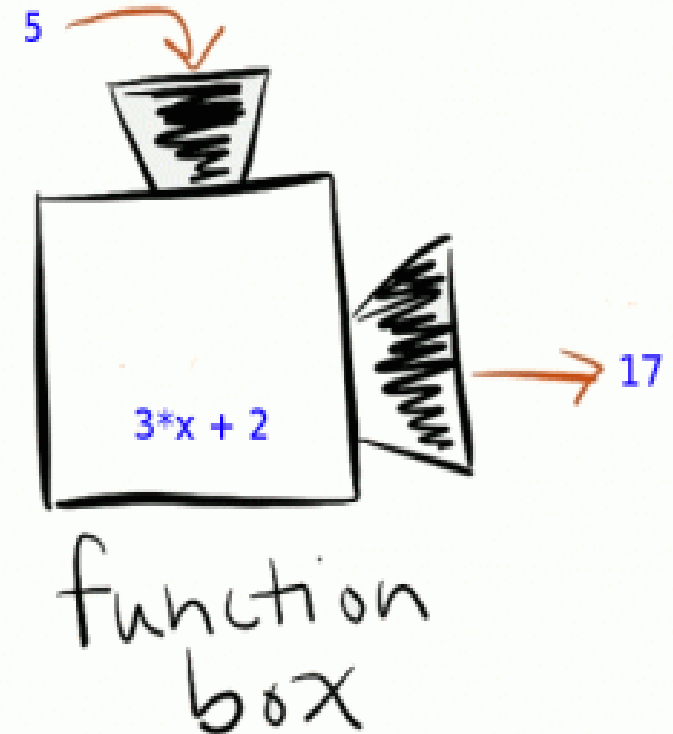


- One of the most important and fundamental constructs for code organization
- A function is written to perform a specific task
- It:
  - Produces output based on input (pure function)
  - Performs a group of actions (procedure)
  - Of both ...
- If you find yourself **typing essentially the same code** more than once, a warning light should go off and you should think:  
“I wonder if I should put this in a function”

Remember those drawings from math class?

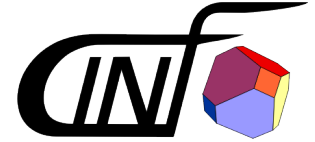


- The inside of the function is isolated
  - Can only use objects that are passed in (almost!)
- Put something in, get something out
- Same same, but a little different



<http://www.teachingepsilon.com/category/functions/>

# Defining functions



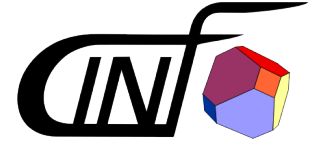
- The def keyword

The function is a block

```
>>> def fib(n):  
...     """Print a Fibonacci series up to n."""  
...     a, b = 0, 1  
...     while a < n:  
...         print(a, end=' ')  
...         a, b = b, a+b  
...  
>>> # Now call the function we just defined:  
... fib(2000)  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597  
  
>>> fib  
<function fib at 10042ed0>
```

Code examples from <https://docs.python.org/3/tutorial/>

# The 'return' statement on functions



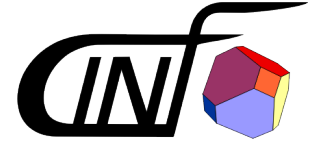
```
>>> def fib2(n):
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)
>>> f100
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Code examples from <https://docs.python.org/3/tutorial/>

- If no return statement is present, it returns 'None'

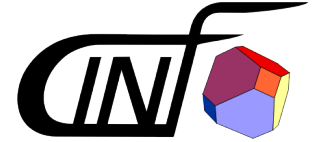
```
>>> def myfunction():
...     pass
...
>>> print myfunction()
None
```

# Default values with keyword arguments

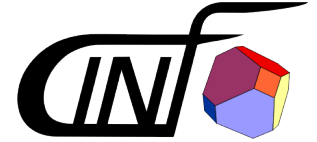


```
>>> def prefixed_print(string, prefix='Now hear this!'):
...     """Prints the string with prefix"""
...     print(prefix, string)
...
>>>
>>>
>>> prefixed_print('Coffee will be served at 10')
Now hear this! Coffee will be served at 10
>>>
>>> prefixed_print('Coffee will be served at 10', prefix='Listen up!')
Listen up! Coffee will be served at 10?
```

# Keyword arguments



```
>>> def prefixed_print(string, prefix='Now hear this!'):
...     """Prints the string with prefix"""
...     print(prefix, string)
...
>>> prefixed_print('Coffee will be served at 10', 'Listen up!')
Listen up! Coffee will be served at 10
>>>
>>> prefixed_print(string='Coffee will be served at 10', prefix='Listen up!')
Listen up! Coffee will be served at 10
>>>
>>> prefixed_print('Coffee will be served at 10', prefix='Listen up!')
Listen up! Coffee will be served at 10
>>>
>>> prefixed_print(string='Coffee will be served at 10', 'Listen up!')
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```



- Use the keywords of keyword arguments if they are non-obvious

```
# What is 4??  
send('Text to be sent', 4)  
  
# Much better  
send('Text to be sent', retries=4)
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear



# Exercise time

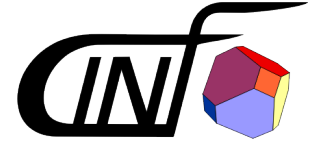
- `extract_from_string_as_function.py`
- Write a function that takes two arguments;
  - a greeting
  - and an optional number of times it is to be repeated, which defaults to 3and returns the result
- Experiment with different calling ways of calling it



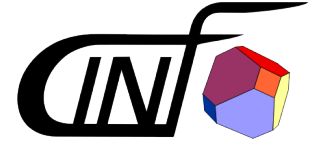
Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1



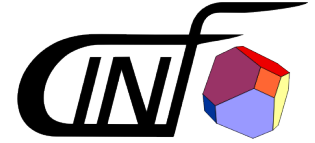
- Comment your code to ease readability
- ALSO if future-you is the only potential reader
  - Future-you will thank you for it (and you never know)
- **Good clean code helps**
  - Descriptive variables names
  - Appropriately sized lines
- But **does not replace comments**



- Find an appropriate level
  - A comment for each logical block
  - And especially important lines or spiffy comments
  - To explain difficult concepts or provide link to documentation

```
# Calculate firing parameters for giant moon laser
coordinates = get_coordinates()
intensities = get_intensities()
firing_parameters = calculate_firing_parameters(coordinates, intensities)

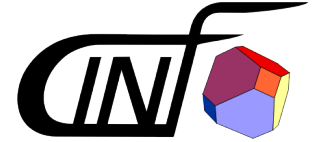
# FIRE!
for parameter_set in firing_parameters:
    adjust_laser_position(parameter_set)
    fire()    # Muhahaha
```



- The first triple quoted string after a function definition or at the beginning of a file is the doc string
- It explains what the functions (or file) does and possible which arguments it takes and what it returns
- **ALL methods deserve a doc string!!!**
- Can also be used for auto-generated documentations



# In-line documentation – doc string



```
def my_fancy_calculation():
    """Calculates the ... by means of ..."""

def fire(parameters):
    """Fires the laser and returns firing success

    Args:
        parameters (dict): A dictionary of laser and shot parameters

    Returns:
        bool: Firing succes
    """
    start_cooling(parameters['power'])
    set_parameters(parameters)
    ramp_up()
    success = fire(parameters['duration'])
    stop_cooling()
    return success
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

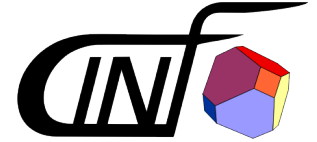
# Exercise time

- Write a doc string for the function you made in `extract_from_string_as_function.py` that documents:
  - What it does in one short line (the first one)
  - What parameters it takes
  - What it returns (if anything)

```
def fire(parameters):  
    """Fires the laser and returns firing success  
  
    Args:  
        parameters (dict): A dictionary of laser and shot parameters  
  
    Returns:  
        bool: Firing succes  
    """  
    pass
```



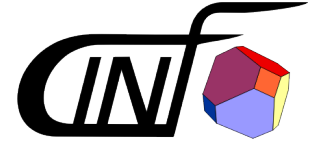
# You really should comment



- It is almost for free, **if**, you do it when you write the code
- You may think:
  - That no-one else will ever read your code
  - That you are a special case, that will easily be able to figure your own code out later and that therefore you do not need to comment
- You are mistaken and it will bite you
- And remember...!

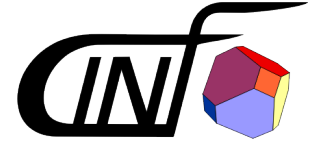
# The Zen of Python

yet again



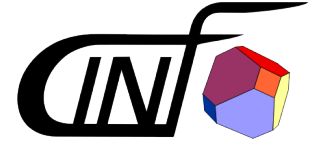
```
>>> import this
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```



- Makes the code easier to read by you
- Makes the code easier to share with others
- Introduces “shortcuts” in identifications of objects and in pin-pointing errors

# Coding style - indentation

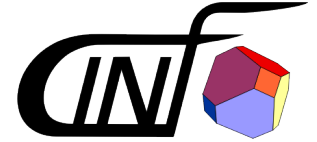


- Because indentation is part of the syntax, extra care needs to be taken
- **Always indent with 4 spaces**
- Check what your editor does and possibly correct it
- *“Thus spake the Lord: Thou shalt indent with four spaces. No more, no less. Four shall be the number of spaces thou shalt indent, and the number of thy indenting shall be four. Eight shalt thou not indent, nor either indent thou two, excepting that thou then proceed to four. Tabs are right out.”*

*-- Georg Brandl*

```
def my function():  
    ....Here starts the code  
    ....if condition:  
        .....This is double indented
```

## Coding style – spaces



- Use spaces around operators and after commas, but not around keyword argument '='

```
a=f(1, 2) + g(3, 4)
a = f(1, 2)+g(3, 4)
a = f(1,2) + g(3,4)
```

```
def myfunc(kwarga = 100):
    pass
```

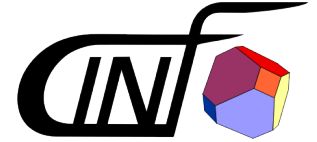
```
myfunc(kwarga = 150)
```

```
a = f(1, 2) + g(3, 4)
```

```
def myfunc(kwarga=100):
    pass
```

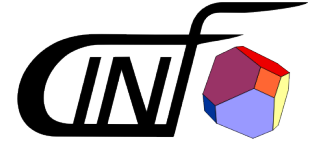
```
myfunc(kwarga=150)
```

## More coding style



- Comments on their own line is preferable
- Wrap lines to not exceed ~~79~~ 90 characters (controversial)

# Naming conventions



- Classes should be named with CamelCase
- Functions and variables should be names with underscore\_separated\_lowercase
- (Module level variables should be UNDERSCORE\_SEPARATED\_UPPERCASE)

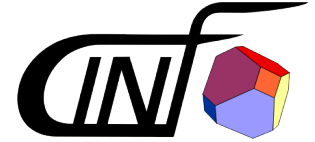
```
PLOT_COLOR = 'blue'

class SetOfMeasurements:
    ...

def plot_set_of_measurements(set_of_measurements):
    for measurement in set_of_measurements:
        plot(measurement, PLOT_COLOR)

measurements_set1 = SetOfMeasurements()
plot_set_of_measurements(measurements_set1)
```

## More on naming



- Verbs in functions
  - `measure()` , `get_parameters()` etc.
- Nouns for variables
  - `measurement`, `plot_parameters`
- Use plural, when there are more than one e.g in a list
  - `measurements`
- Think about naming, it will make it easier for yourself to figure out what you are working on
- `'output'`, `'input'`, `'content'`, `'text'`, `'string'`, `'constant'`, `'limit'` etc. are not necessarily very helpful
- `'x'`, `'y'`, `'t'` etc. are o.k. locally, as long as it isn't ambiguous



# Exercise time

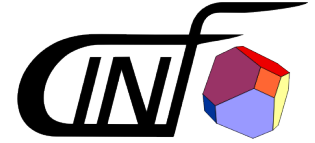
- For the function in `extract_from_string_as_function.py` clean it up with respect to code style focusing on
  - Indentation
  - Spaces
  - Naming patterns for different items of code
  - Helpful naming of functions and variables
- Rejoice, as you notice how much easier your code is to read



Center for Individual Nanoparticle Functionality

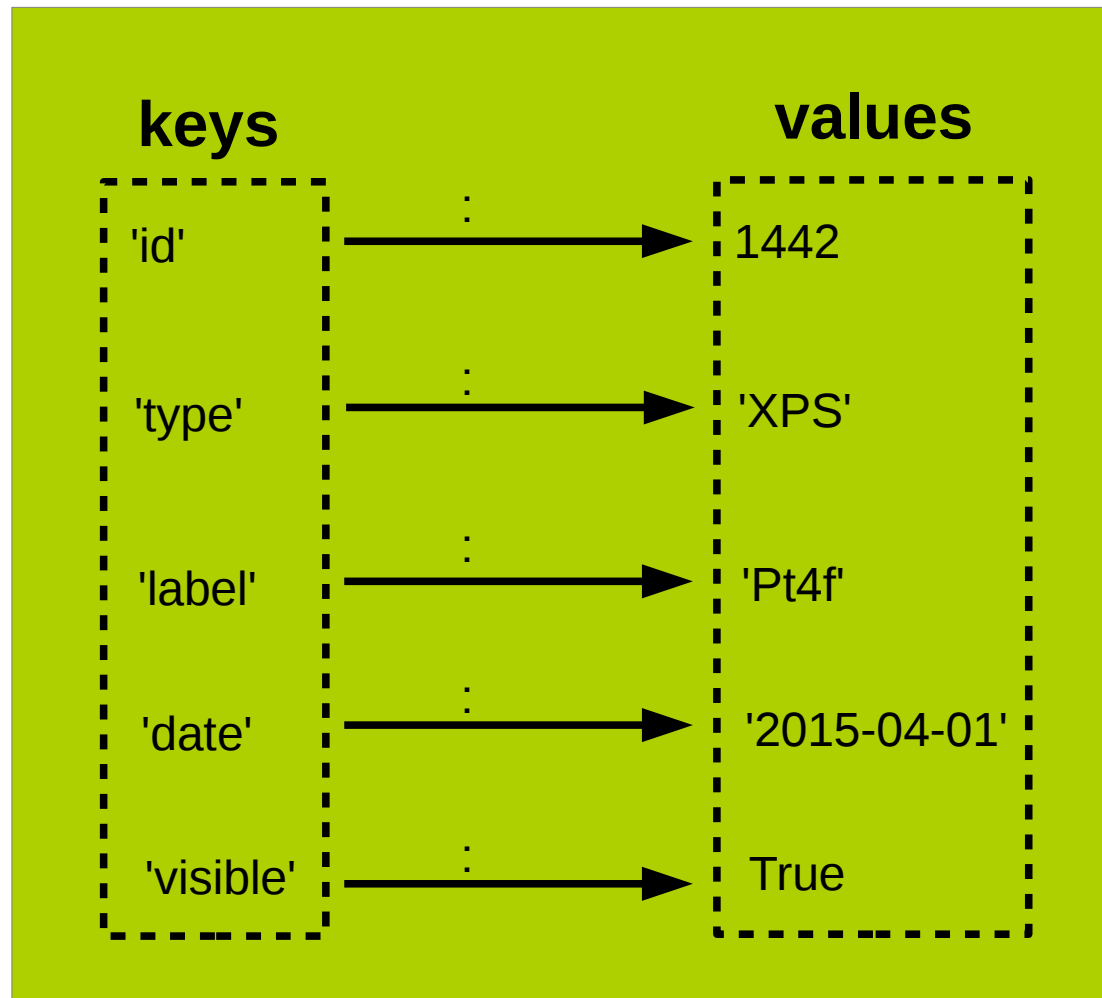
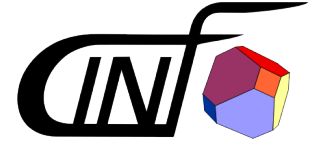
- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1

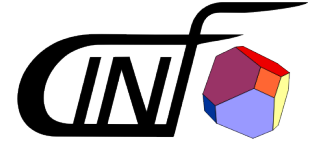



- An unordered set of key: value pairs
- In other languages sometimes referred to as an *associative array* or a *hash map*
- Indexed by keys
- Keys can be any completely immutable object
  - I.e. an immutable object that contains only immutable objects
- **Incredibly** useful for storing values e.g. with names

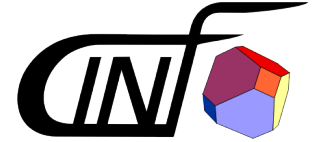
# dict



## More about dictionaries

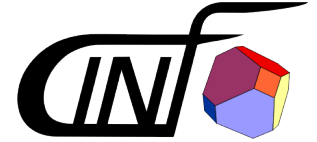


- The key needs to be hashable
  - You cannot use e.g. lists or dicts
- Dicts are incredibly fast even when they are large, but it comes at the expense of space 
- If you need only keys, you need a **set**



- dicts has several methods that are useful for looping:
  - **keys()** mostly not necessary, dict is seq of keys
  - **values()**
  - **items()**
- And lots of other useful methods. Have a look at them with help.

# Dict examples



```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}

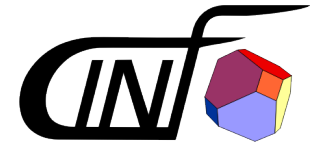
>>> tel['jack']
4098

>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}

>>> tel.keys()
['guido', 'irv', 'jack']

>>> 'guido' in tel
True
```

Code examples from <https://docs.python.org/3/tutorial/>



## More dict examples

```
>>> # Calling dict on a sequence of two value sequences
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}

>>> # Dictionary comprehensions
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Code examples from <https://docs.python.org/3/tutorial/>

```
>>> # Dictionary comprehensions: reversing a dict
>>> my_dict = {'first_key': 5, 'second_key': 10}
>>> my_dict
{'second_key': 10, 'first_key': 5}
>>> reversed_dict = {value: key for key, value in my_dict.items()}
>>> reversed_dict
{10: 'second_key', 5: 'first_key'}
```



# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

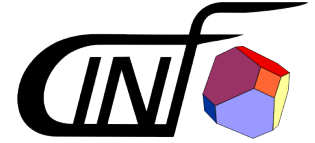
- `extract_key_value_pairs_from_string.py`
- The BIG exercise webscraping



Center for Individual Nanoparticle Functionality

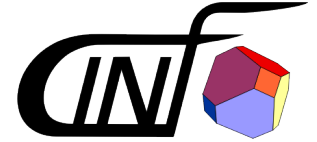
- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- [Looping techniques](#)
- List comprehensions
- Imports
- Exceptions (brief)

# Python part1



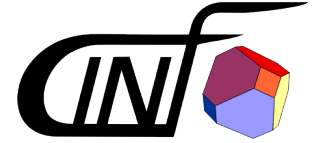
- Incredibly useful programming patterns used to:
  - Avoid manual indexing
  - Increase readability
  - Express complex algorithms simpler
- A lot of effort has been made to ensure that you do not need to index manually
  - Because of-by-1 errors are common

# Looping techniques



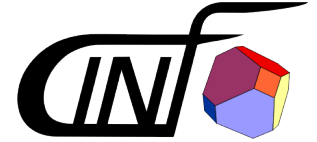
- Whenever possible, you should always use a looping technique
- If you are looping over the numbers in range only to index something, you are (most likely) doing it wrong!

# Just don't do it



```
>>> colors = ['red', 'green', 'blue']
>>> for index in range(len(colors)):
...     print(colors[index])
...
red
green
blue
>>> # Noooooooooooooooooooooo!
```

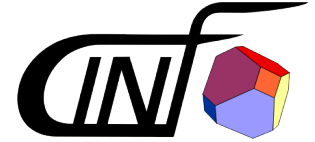
```
>>> colors = ['red', 'green', 'blue']
>>> for color in colors:
...     print(color)
...
red
green
blue
### # Delicious, readable and much less error prone
```



- But what if I need the index or if I need to loop over 2 lists?
- **enumerate()** - enumerates the object
- **zip()** - zips together two or more lists
- Lots more tools like this, to modify iteration in **itertools** module



# Looping techniques



```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe

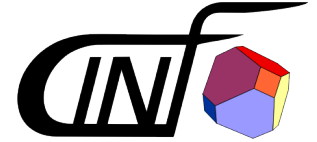
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.

>>> for i in reversed(range(1,10,2)):
...     print(i)
...
9
7
5
3
1
```

Code examples from <https://docs.python.org/3/tutorial/>



# Looping techniques with dicts



```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Code examples from <https://docs.python.org/3/tutorial/>

```
>>> # Or just the keys
... for knight in knights: # Note, .keys() is not necessary
...     print(knight)
...
gallahad
Robin

>>> # Or just the values
... for description in knights.values():
...     print(description)
...
the pure
the brave
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- Make a list of colors and print out the color and its index by use of enumerate
- Print out the content of two lists pairwise by use of zip

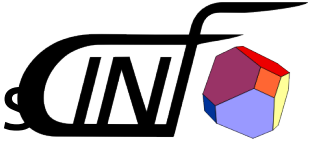


Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- [List comprehensions](#)
- Imports
- Exceptions (brief)

# Python part1

# Functional programming with sequences



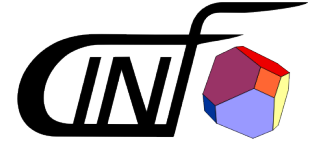
- `filter(function, sequence)`
  - Returns a sequence of those items in sequence where `function(item)` is true
- `map(function, sequence0, sequence1, ...)`
  - Returns a sequence of `function(item)` for each item in sequence
  - Can be called with `n` sequences provided function takes `n` arguments
- `reduce(function, sequence)`
  - Returns a single argument formed by calling function on the first two arguments and then on the result of that and the next argument and so on

## Warning

### Slightly opinionated content

- Now, please forget everything about:
  - `map`, `filter`, `reduce` and `lambda`
- ... except for the fact that they exist
- There is a better way!

# List comprehensions



- Forming a list by iterating over another list is such a common operation that a programming shortcut has been made

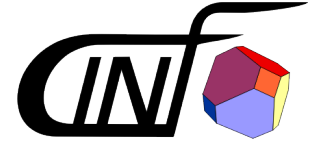
```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> ### Use a list comprehension to do the same
>>> squares = [x**2 for x in range(10)]
```

```
>>> ### Can also be achieved with a map, but the list comprehension is much
>>> ### more readable
>>> squares = map(lambda x: x**2, range(10))
```

Code examples from <https://docs.python.org/3/tutorial/>

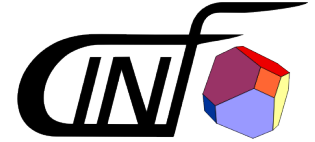
# Can contain a condition



```
>>> [x for x in range(10) if x % 2 == 0]  
[0, 2, 4, 6, 8]
```



# More list comprehension examples



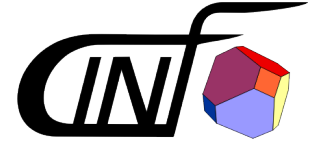
```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]

>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]

>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
```

Code examples from <https://docs.python.org/3/tutorial/>

# More list comprehension examples



```
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

```
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

```
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Code examples from <https://docs.python.org/3/tutorial/>

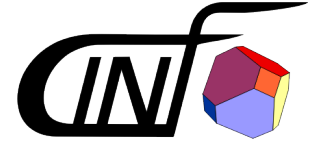
# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

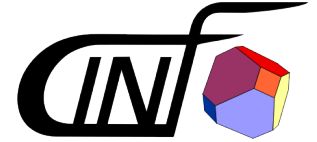
- Use the **append** and **pop** methods to add and pop of elements in a list
- Make a list comprehension that calculates squares of the elements in a list

# Tuples and sequences



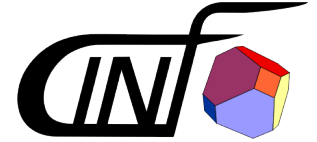
- Notice that `strs` and `lists` have a lot of functionality in common:
  - Indexing, slicing, addition, multiplication etc.
- That is because they are both **sequences**
- A data type that performs certain operations in a specific way
  - <https://docs.python.org/2/library/stdtypes.html#typeseq>
- Several sequences: `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange`

# Tuples



- Created with parentheses (1, 2, 3)
- Only shown here for information
- Use lists for most applications
- Think of them as immutable (unchangeable) lists
- Tuples are used a lot behind the scenes for multiple return values from functions

# Scientists summary on containers



- lists and dicts really are the bread and butter for organization of datasets and metadata
  - Use them and get used to them
- (Not for data, more about that on day 2)
- Branch out into tuples and sets if necessary
- There are plenty more specialized container types that you can search for if you need them:
  - Named tuple, deque, ordered dictionary, default dictionary, counter ...

Now we switch gears  
and move on  
from general programming techniques  
to more organizational subjects

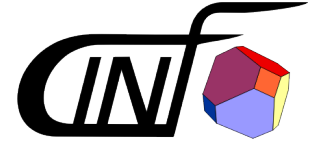




Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- [Imports](#)
- Exceptions (brief)

# Python part1



- A module is simply a file (not case for all lang.)
- If the file is in your import path (the current folder always is) you can import from the top level:
  - Variables
  - Functions
- Which will make those objects available to you where you import them

# Module example

Content of the file fibo.py

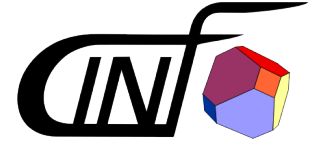
```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Code examples from <https://docs.python.org/3/tutorial/>

# Module example



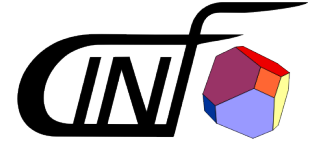
```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> # Importing everything from a module into your local namespace is
>>> # considered a bad practice
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Code examples from <https://docs.python.org/3/tutorial/>

# Module example

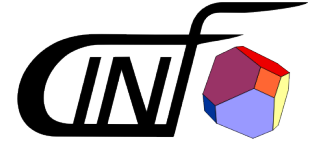


```
>>> from fibo import fib as my_fancy_fibonacci_function
>>> my_fancy_fibonacci_function(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

# Exercise time

- Write a function to a file
- Import that function from a script using some of the different importing patterns we just discussed
  - `import module`
  - `import module as new_name_for_module`
  - `from module import function_name`
  - `from module import *`
  - `from module import func_name as new_func_name`

# Executing a module

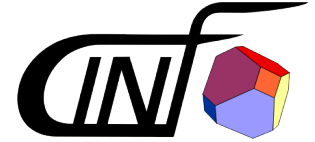


- On import every statement in the module will get executed
- If you have statements (e.g. test code) that you only wish to execute if the module itself is executed, put it in an if-statement:
- The `__name__` of the module that was executed is always `'__main__'`

```
# Module code

if __name__ == "__main__":
    # Module test code'
    pass
```

# Module search path



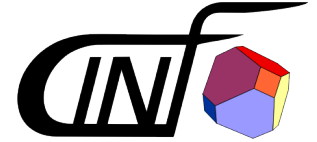
- The list of paths that Python will look for modules in is available in `sys.path`
- And you can modify it, if you wish to add a folder with common code

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old', ...]

>>> sys.path.insert(1, '/my/dir/with/fancy/code')
>>> sys.path
['', '/my/dir/with/fancy/code', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu', '/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old', ...]
```



# Packages

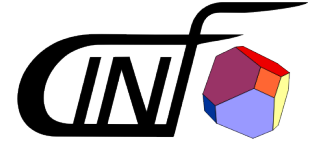


- You can make files in subdirectories of your search path importable by turning them into a package
- By adding an empty `__init__.py` file to the folder

```
# Directory listing
my_math/
  __init__.py
  fibo.py

# Code
from my_math.fibo import fib
```

# The dir function



- Can be used to figure out which names a module (or class) defines

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
```

# Exercise time

- Investigate which names are defined in the 'collections' module
  - The collections module, among other things, contain more specialized container types

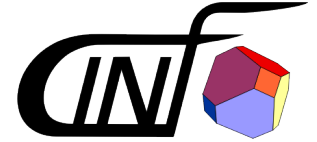


Center for Individual Nanoparticle Functionality

- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- Looping techniques
- List comprehensions
- Imports
- [Exceptions \(brief\)](#)

# Python part1

# Exceptions can be caught



- Everything else than syntax errors can be **caught** and handled
- May be useful if code sometimes fails predictively
- Ask forgiveness instead of permission
- Not terribly useful for data treatment, so will not cover further



```
try:  
    title = mymetadata['title']  
except KeyError:  
    title = 'No title'
```

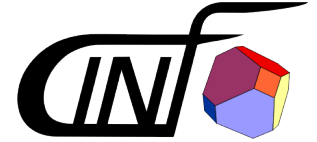


Center for Individual Nanoparticle Functionality

- Summary

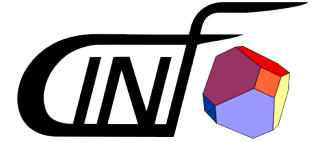
# Python part1

# Summary – About Python



- Python is good for scientist due to:
  - Fast and easy development
  - Code re-use
  - Large scientific community (lots of packages)
  - Always available (also after a change of job)

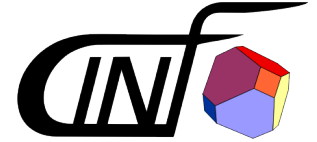
## Summary – Common concepts



- In Python you will find all the same programming constructs that you know
  - Variables
  - Types (int, float, string)
  - Containers (lists, dicts)
  - Control flow (if, for, while)
  - Functions
  - Importing of functionality
  - Exceptions (Errors)
- Familiar syntax

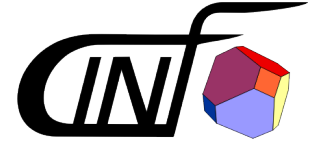


## Summary – About containers



- Get to know you basic container types:
  - Lists
  - Dicts
- They will be at the heart of everything you do
- Learn to loop over these containers in **the right way**
  - Manual indexing is error prone, and the Python developers has made a large effort to make sure you don't have to

## Summary – About types



- Dynamic typing is a blessing and a curse
- Makes many things easier, but includes the potential for errors
- Proper descriptive naming of variables and doc strings is your best defense

THE END

THE END

THE END

The End

Extra slides

# Errors and exceptions



# Syntax errors

- Produces by the parser, if it encounters something that it does not understand
- Produced before the program starts running

```
>>> while True print 'Hello world'  
File "<stdin>", line 1, in ?  
    while True print 'Hello world'  
                    ^  
SyntaxError: invalid syntax
```

Code examples from <https://docs.python.org/3/tutorial/>

- Preceding the little arrow is the first item that the parser does not understand

# General exceptions

- Raised when something unexpected happens **while** the program is running
  - Execution stops at the point where the exception is raised
  - It travels up the call stack (the order in which functions are called)
  - Either until it reaches the top of it is intercepted
- Exceptions need not be an error (catastrophe)
- Code can fail in an expected way

# Exception examples

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Code examples from <https://docs.python.org/3/tutorial/>

# Exceptions

- Exception can be caught
- `try` and `except` keywords
- Will continue execution of the program at the point where it was caught
- But it should **only** be done for **expected** failures
- I.e. exceptions should be caught as narrowly as possibly

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        +-- BufferError
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- EnvironmentError
            +-- IOError
            +-- OSError
                +-- WindowsError (Windows)
                +-- VMSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            +-- NotImplementedError
        +-- SyntaxError
            +-- IndentationError
            +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            +-- UnicodeError
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError
    +-- Warning
        +-- DeprecationWarning
    ...
```

# The Zen of Python

Our old friend returns

```
>>> import this
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

# Exception catching examples

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
```

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Code examples from <https://docs.python.org/3/tutorial/>

# Catching several exceptions

```
>>> try:  
...     1/0  
... except (ZeroDivisionError, TypeError) as e:  
...     print e
```

Code examples from <https://docs.python.org/3/tutorial/>

# Raising and defining exceptions

- Exception can be raised with the raise keyword
  - The sole argument is an instance of the exception to be raised
- New exceptions can be defined by deriving from the Exception class
  - But if there is a standard error that covers your case, you should use that,
  - Unless you really need to differentiate
- What? Instance, deriving class?
  - We will get to that in the next section



# Defining and raising

```
>>> # Defining an exception
>>> class MyFancyException(Exception):
...     pass
...
>>> # Raising it
>>> raise MyFancyException('The error message for my fancy exception')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyFancyException: The error message for my fancy exception

>>> # Raising a standard exception
... raise TypeError('You gave me the wrong type!')
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: You gave me the wrong type!
```

# try-except-else-finally, oh my!

- Logic concerning exception catching and clean up can become complicated
- Built-in constructs to help
  - **try** is for (the) line(s) that might fail
  - **except** performs actions in response to failure
  - **else** performs more actions, that are not expected to raise error, after success
  - **finally** is for clean-up in any case, and is also executed if there are uncaught exceptions



<http://forums.udacity.com/questions/5008438/try-finally>

# Try-except-else-finally example

```
>>> def divide(x, y):  
...     try:  
...         result = x / y  
...     except ZeroDivisionError:  
...         print "division by zero!"  
...     else:  
...         print "result is", result  
...     finally:  
...         print "executing finally clause"  
...
```

```
>>> divide(2, 1)  
result is 2  
executing finally clause
```

```
>>> divide(2, 0)  
division by zero!  
executing finally clause
```

```
>>> divide("2", "1")  
executing finally clause  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
  File "<stdin>", line 3, in divide  
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- FIXME add exception exercise

# Lists are ideal for simple stacks

## Last In First Out (LIFO)

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Code examples from <https://docs.python.org/3/tutorial/>

# For list based queues use deque First In First Out (FIFO)

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Code examples from <https://docs.python.org/3/tutorial/>

# Tuple examples

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Code examples from <https://docs.python.org/3/tutorial/>



# Creating tuples with 0 or 1 element

- Because parentheses are used for other purposes, special syntax is needed to create a tuple with 0 or 1 element

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Code examples from <https://docs.python.org/3/tutorial/>

# When to use tuples and lists

- Tuples are immutable and are usually used for heterogeneous sequence accessed by indexing or unpacking
  - ('Kenneth', 'Nielsen', 183)
- Lists are mutable and are usually used for homogeneous sequences accessed by indexing
  - [1, 2, 3]
  - [('Kenneth', 'Nielsen', 183), ('Robert', 'Jensen', 186)]

# Tuple packing and unpacking

- Used to automatically put several values in a tuple
- Or to automatically pull all values out of a tuple and into e.g. variables
- Incredibly useful for:
  - Temporary storage of several values
  - Multiple return values from functions
  - Simultaneous iterative updating of values

```
>>> # Multiple values separated by comma are automatically turned into a tuple
>>> # Packing
>>> my_values = 1, 2, 3
>>> my_values
(1, 2, 3)

>>> # Automatic assignment to the correct number of comma separated variables
>>> # Unpacking
>>> value0, value1, value2 = my_values
>>> value0
1
>>> value2
3

>>> # Simultaneous updating of several value done right
>>> a, b, c = 1, 2, 3
>>> a, b, c = 10, a, b
>>> a, b, c
(10, 1, 2)

>>> # Multiple return values from functions
>>> def my_function():
...     return 1, 2
...
>>> return_value = my_function()
>>> return_value
(1, 2)
>>> return_value0, return_value1 = my_function()
>>> return_value0
1
>>> return_value1
2
```

# Type along

- We'll experiment with the covered material
- Open up your interpreter and type along
- **Ask** questions if something is not clear

# Exercise time

- FIXME tuple exercises

Classes  
and  
object-oriented (OO) programming  
  
aka  
“The good stuff”

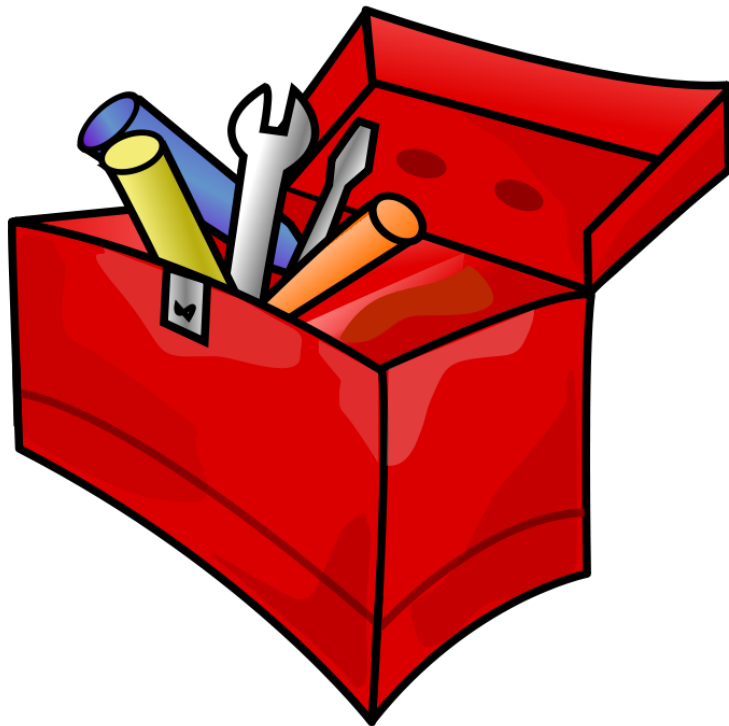
# Raising the abstraction level (a bit)

- Forget all about the details for a moment
- Imagine you are in your happy place
- Let the tranquility there fill you up
- OO is considered slightly advanced and maybe even a little hard to learn
- It involves a few new and fundamentally different aspects
- But it can be incredibly practical



# Taking a step back

So far we have had our heads buried deep in the toolbox!



<https://openclipart.org>

Now lets talk technique



<http://www.oncoloring.com/spongebob-squarepants-coloring-pages.html>

What 2 things does a computer program (computing) consist of?

# What does a computer program (computing) consist of?

- Data

```
"measurement"  
[0, 1, 2, 3]  
[1.23, 2.34, 3.45, 4.56]  
"settings"  
100.0  
47
```

- Algorithms that manipulate that data

## INEFFECTIVE SORTS

```
DEFINE FASTBOGOSORT(LIST):  
  // AN OPTIMIZED BOGOSORT  
  // RUNS IN O(N LOG N)  
  FOR N FROM 1 TO LOG(LENGTH(LIST)):  
    SHUFFLE(LIST):  
    IF ISORTED(LIST):  
      RETURN LIST  
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE PANICSORT(LIST):  
  IF ISORTED(LIST):  
    RETURN LIST  
  FOR N FROM 1 TO 10000:  
    PIVOT = RANDOM(0, LENGTH(LIST))  
    LIST = LIST[PIVOT:] + LIST[:PIVOT]  
    IF ISORTED(LIST):  
      RETURN LIST  
  IF ISORTED(LIST):  
    RETURN LIST  
  IF ISORTED(LIST): // THIS CAN'T BE HAPPENING  
    RETURN LIST  
  IF ISORTED(LIST): // COME ON COME ON  
    RETURN LIST  
  // OH JEEZ  
  // I'M GONNA BE IN SO MUCH TROUBLE  
  LIST = []  
  SYSTEM("SHUTDOWN -H +5")  
  SYSTEM("RM -RF ./")  
  SYSTEM("RM -RF ~/*")  
  SYSTEM("RM -RF /")  
  SYSTEM("RD /S /Q C:\*") // PORTABILITY  
  RETURN [1, 2, 3, 4, 5]
```

# Programming paradigms

## “very simplified”

- Among other things ..!
- How do we organize the data and the algorithms that manipulate that data
- Different focuses
  - Program efficiency (how fast does it run)
  - Programming efficiency (how fast can I make the program)
  - Learning curve (for the paradigm)
  - Readability
  - Abstraction level
  - Error prone
  - De-bug-ability
  - Maintainability (can it be altered, used and read by others)

# Data and algorithms separate

```
# Functions for measurement sets
def ms_get_measurement_set():
    return []

def ms_add_measurement(measurement_set, measurement):
    measurement_set.append(measurement)

def ms_get_measurement_by_name(measurement_set, name):
    for measurement in measurement_set:
        if measurement['name'] == name:
            return measurement

# Functions for measurements
def m_create_measurement(name, points):
    return {'name': name, 'points': points}

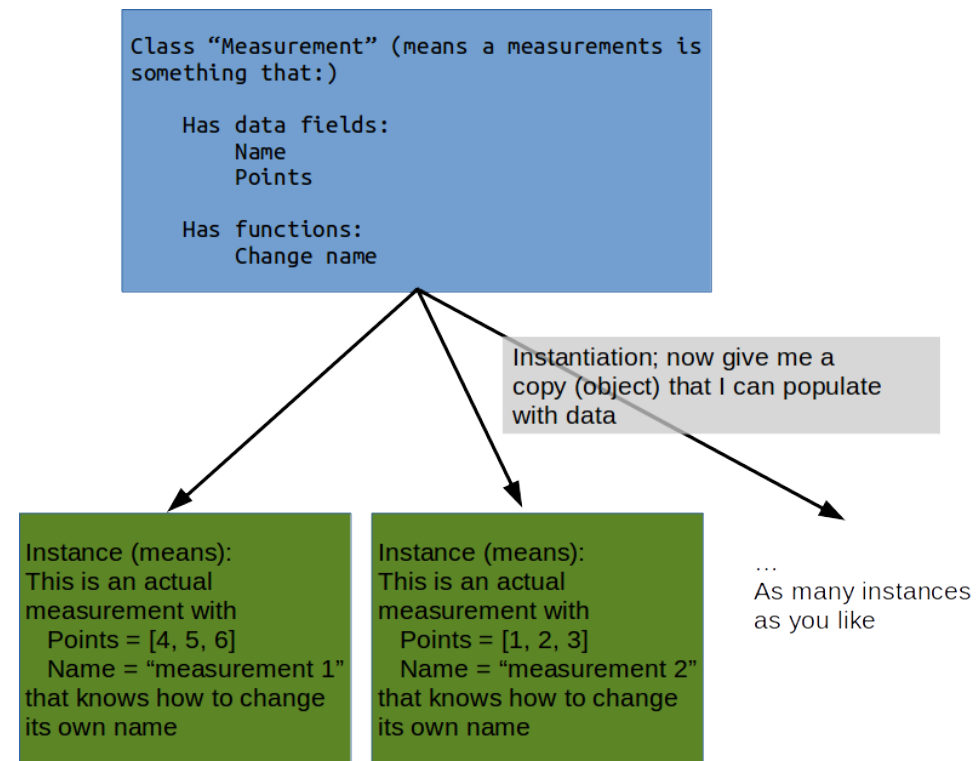
def m_change_name(measurement, new_name):
    measurement['name'] = new_name
```

- The data is contained at the highest level
- Functions will need some sort of prefix:
  - To prevent name conflicts
  - To figure out which functions act on which kind of data

Demo

# Object oriented programming

- Stores algorithms **together with** the data they manipulate in **objects**
- Objects are created as instances (copies) of classes (definitions)
- A **class** contains:
  - Definitions of what data it can contain (not the data itself)
  - The algorithms that manipulate that data
- A class is created with the **class** keyword



# Object oriented

```
class Measurement(object):  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
class MeasurementSet(object):  
    def __init__(self, measurements=None):  
        if measurements is not None:  
            self.measurements = list(measurements)  
        else:  
            self.measurements = []  
  
    def add_measurement(self, measurement):  
        self.measurements.append(measurement)  
  
    def find_measurement(self, name):  
        for measurement in self.measurements:  
            if measurement.name == name:  
                return measurement
```

- Data in identifiable objects
- Functions (methods really) are attached to the object

Demo

# Class dissection – name and bases

- The `class` keyword
- The name of the class
  - Traditionally in CamelCase
- `()` with parents
  - We will get to inheritance later
  - If nothing else, just put `object` here
- A colon

```
class Measurement(object):  
  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)
```



# Class dissection – class content

- Indent to indicate class content

```
class Measurement(object):  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)
```

# Class dissection – methods

- Methods are defined like functions (**def**)
- Always **self** as the first argument\*
- **self** is automatically added when you call a method
- Remaining argument(s)
  - What you call it with
  - Available in that method

```
class Measurement(object):  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
measurement = Measurement('My meas', [1, 2, 3])  
# Notice only name argument  
Measurement.change_name('New name')
```

# Class dissection - `__init__`

- Initializer method
- Automatically called with instantiation arguments
- Data is assigned to data fields on the object
  - Remember arguments to methods are only available inside the method

```
class Measurement(object):  
  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
Measurement('My measurement name', [1, 2, 3])  
# Puts 'My measurement name' into name  
# and [1, 2, 3] into points in __init__
```

# Class dissection – accessing properties data and methods

- The . operator
- .name is a data field
- .change\_name is a method

```
class Measurement(object):  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
meas = Measurement('The name', [1, 2, 3])  
print meas.name  
The name  
meas.change_name('New name')  
print meas.name  
New name
```

# The main points

- To teach you the basics of OO as it will be useful to understand how you use library functionality
- When you understand the idea, you may find it useful for organizing larger data treatment programs ...
- ... and for making data less anonymous

# OO in usage

```
from collections import Counter

my_counter = Counter()

for item in [4, 7, 4, 2, 4, 9, 9]:
    my_counter[item] += 1

print my_counter

## Outputs
Counter({4: 3, 9: 2, 2: 1, 7: 1})
```

# OO for organization

```
class Measurement(object):  
    pass
```

```
class MeasurementSet(object):  
    pass
```

```
class DataImporter(object):  
    pass
```

```
class DataTreatment(object):  
    pass
```

# OO for less anonymous data

- For data (and meta data) contained purely in standard containers it can be difficult to identify the type
- OO fixes that, by always having the class present, which **is** the type



# Exercise time

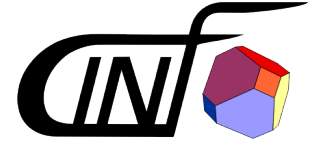
- FIXME add OO exercise



Center for Individual Nanoparticle Functionality

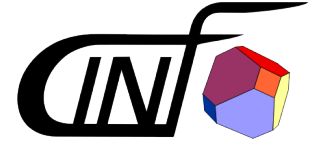
- Variables, types, comparisons
- Strings
- Lists
- Loops
- if-elif-else
- Functions
- (Comments and docstrings)
- Dicts
- List comprehensions
- Looping techniques
- Imports
- Exceptions (brief)
- **Classes (object oriented programming)**

# Python part1



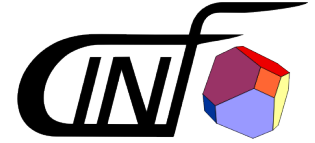
- Classes are what defines types
- They describe:
  - What data they can contain
  - What methods they have
  - If they can be changed
- Will not cover writing classes
- Will have a look at them, to understand how to use them

# Class dissection – name and bases



- The `class` keyword
- The name of the class
  - Traditionally in CamelCase
- `()` with parents
  - We will get to inheritance later
  - If nothing else, just put `object` here
- A colon

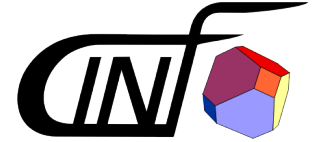
```
class Measurement(object):  
  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)
```



- Indent to indicate class content

```
class Measurement(object):  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)
```

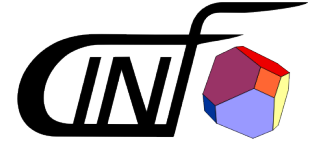
# Class dissection – methods



- Methods are defined like functions (**def**)
- Always **self** as the first argument\*
- **self** is automatically added when you call a method
- Remaining argument(s)
  - What you call it with
  - Available in that method

```
class Measurement(object):  
  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
measurement = Measurement('My meas', [1, 2, 3])  
# Notice only name argument  
Measurement.change_name('New name')
```

# Class dissection - `__init__`

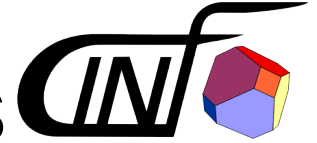


- Initializer method
- Automatically called with instantiation arguments
- Data is assigned to data fields on the object
  - Remember arguments to methods are only available inside the method

```
class Measurement(object):  
  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
Measurement('My measurement name', [1, 2, 3])  
# Puts 'My measurement name' into name  
# and [1, 2, 3] into points in __init__
```

# Class dissection

## accessing properties data and methods

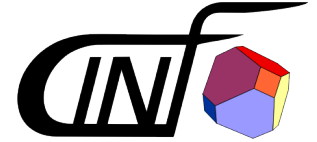


- The . operator
- .name is a data field
- .change\_name is a method

```
class Measurement(object):  
  
    def __init__(self, name, points):  
        self.name = '##_{}'.format(name)  
        self.points = points  
  
    def change_name(self, new_name):  
        self.name = '##_{}'.format(new_name)  
  
meas = Measurement('The name', [1, 2, 3])  
print meas.name  
The name  
meas.change_name('New name')  
print meas.name  
New name
```

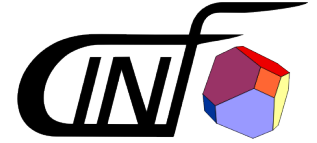


## The main points



- To teach you the basics of OO as it will be useful to understand how you use library functionality
- When you understand the idea, you may find it useful for organizing larger data treatment programs ...
- ... and for making data less anonymous

## OO in usage



```
from collections import Counter

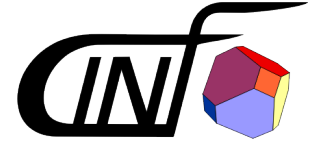
my_counter = Counter()

for item in [4, 7, 4, 2, 4, 9, 9]:
    my_counter[item] += 1

print my_counter

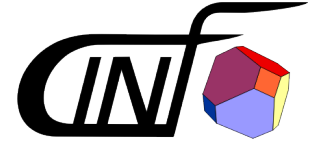
## Outputs
Counter({4: 3, 9: 2, 2: 1, 7: 1})
```

# OO for organization



```
class Measurement(object):  
    pass  
  
class MeasurementSet(object):  
    pass  
  
class DataImporter(object):  
    pass  
  
class DataTreatment(object):  
    pass
```

## OO for less anonymous data



- For data (and meta data) contained purely in standard containers it can be difficult to identify the type
- OO fixes that, by always having the class present, which **is** the type

# Exercise time

- Import OrderedDict from collections
- Make an instance of it
- loop over and print the key, value pairs:  
  .items()
- What happens?