# Primitive Datatypes & Recursion in Python

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

In python

every value has a datatype,

But you do not need to declare it.

How does that work?

Based on each variable's assignment, python figures out what type it is and keeps tracks of that internally.

# Python's Native Datatypes
## Boolean



### Python provides two constants

- **True** and **False**

### Operations on Booleans

| logic operators |
|---|
| `and or not` |
| logical and, or and negation respectively |
| **relational operators** |
| `== !=` |
| equal and not equal to operators |
| `< > <= >=` |
| less than, greater than, less than or equal to and greater than or equal to operators |

### Note that python allows chains of comparisons

```
[17:42]cazzola@hymir:~/esercizi-pa>python3
>>> x=3
>>> 1<x<=5
True
```

### Two kinds of numbers: integers and floats

- no class declaration to distinguish them
- they can be distinguished by the presence/absence of the decimal point.

```
[15:26]cazzola@hymir:~/esercizi-pa>python3
>>> type(1)
<class 'int'>
>>> isinstance(1, int)
True
>>> 1+1
2
>>> 1+1.0
2.0
>>> type(2.0)
<class 'float'>
>>>
[15:27]cazzola@hymir:~/esercizi-pa>
```

- `type()` function provides the type of any value or variable;
- `isinstance()` checks if a value or variable is of a given type;
- adding an int to an int yields another int but adding it to a float yields a float.

## Coercion & Size

- **int**() function truncates a float to an integer;
- **float**() function promotes an integer to a float;
- integers can be arbitrarily large;
- floats are accurate to 15 decimal places.

## Operators (just a few)

| operators |
|---|
| `+ -` sum and subtraction operators |
| `* **` product and power of operators, e.g., `2 ** 5 = 32` |
| `/ // %` floating point and integer division and remainder operators respectively |

A python list looks very closely to an array

- direct access to the members through [];

```
[12:29]cazzola@hymir:~/esercizi-pa>python3
>>> a_list = ['1', 1, 'a', 'example']
>>> type(a_list)
<class 'list'>
>>> a_list
['1', 1, 'a', 'example']
>>> a_list[0]
'1'
>>> a_list[-2]
'a'
[12:30]cazzola@hymir:~/esercizi-pa>
```

But

- negative numbers give access to the members backwards, i.e., `a_list[-2] == a_list[4-2] == a_list[2]`;
- the list is not fixed in size;
- the members are not homogeneous

A slice of a list can be yielded by the [:] operator and specifying the position of the first item you want in the slice and of the first you want to exclude

```
[13:02]cazzola@hymir:~/esercizi-pa>python3
>>> a_list=[1, 2, 3, 4, 5]
>>> a_list[1:3]
[2, 3]
>>> a_list[:-2]
[1, 2, 3]
>>> a_list[2:]
[3, 4, 5]
[13:03]cazzola@hymir:~/esercizi-pa>
```

Note that omitting one of the two indexes you get respectively the first and the last item in the list.

```
[14:13]cazzola@hymir:~/esercizi-pa>python3
>>> a_list = ['a']
>>> a_list = a_list+[2.0, 3]
>>> a_list
['a', 2.0, 3]
>>> a_list.append(True)
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['four', 'Ω'])
>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']
>>> a_list.insert(0, 'α')
>>> a_list
['α', 'a', 2.0, 3, True, 'four', 'Ω']
```

## Four ways

- \+ operator concatenates two lists;
- `append()` method appends an item to the end of the list;
- `extend()` method appends a list to the end of the list;
- `insert()` method inserts an item at the given position.

## Note

```
>>> a_list.append([1, 2, 3, 4, 5])
>>> a_list
['α', 'a', 2.0, 3, True, 'four', 'Ω', [1, 2, 3, 4, 5]]
```

You can check if an element is in the list

```
>>> a_list = [1, 'c', True, 3.14, 'cazzolaw', 3.14]
>>> 3.14  in a_list
True
```

Count the number of occurrences

```
>>> a_list.count(3.14)
2
```

Look for an item position

```
>>> a_list.index(3.14)
3
```

Note that

```
>>> a_list.index('walter')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError:  list.index(x):  x not in list
```

Elements can be removed by

– position

```
>>> a_list = [1, 'c', True, 3.14, 'cazzolaw', 3.14]
>>> del a_list[2]
>>> a_list
[1, 'c', 3.14, 'cazzolaw', 3.14]
```

– value

```
>>> a_list.remove(3.14)
>>> a_list
[1, 'c', 'cazzolaw', 3.14]
```

In both cases the list is compacted to fill the gap.

Tuples are immutable lists.

```
>>> a_tuple = (1, 'c', True, 3.14, 'cazzolaw', 3.14)
>>> a_tuple
(1, 'c', True, 3.14, 'cazzolaw', 3.14)
>>> type(a_tuple)
<class 'tuple'>
```

### As a list

- parenthesis instead of square brackets;
- ordered set with direct access to the elements through the position;
- negative indexes count backward;
- slicing.

### On the contrary

- no `append()`, `extend()`, `insert()`, `remove()` and so on

# Python's Native Datatypes
## Tuples (Cont'd)



### Multiple Assignments

Tuple can be used for multiple assignments and to return multiple values.

```
>>> a_tuple = (1, 2)
>>> (a,b) = a_tuple
>>> a
1
>>> b
2
```

### Benefits

- tuples are faster than lists
- tuples are safer than lists
- tuples can be used as keys for dictionaries.

# Python's Native Datatypes
## Sets

Sets are unordered "bags" of unique values.

```
>>> a_set = {1, 2}
>>> a_set
{1, 2}
>>> len(a_set)
2
>>> b_set = set()
>>> b_set
set()      ''' empty set '''
```

A set can be created out of a list

```
>>> a_list = [1, 'a', 3.14, "a string"]
>>> a_set = set(a_list)
>>> a_set
{'a', 1, 'a string', 3.14}
```

## Adding elements to a set

```
>>> a_set = set()
>>> a_set.add(7)
>>> a_set.add(3)
>>> a_set
{3, 7}
>>> a_set.add(7)
>>> a_set
{3, 7}
```

– sets do not admit duplicates so to add a value twice has no effects.

## Union of sets

```
>>> b_set = {3, 5, 3.14, 1, 7}
>>> a_set.update(b_set)
>>> a_set
{1, 3, 5, 7, 3.14}
```

## Removing elements from a set

```
>>> a_set = {1, 2, 3, 5, 7, 11, 13, 17, 23}
>>> a_set.remove(1)
>>> a_set
{2, 3, 5, 7, 11, 13, 17, 23}
>>> a_set.remove(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
>>> a_set.discard(4)
>>> a_set
{2, 3, 5, 7, 11, 13, 17, 23}
>>> a_set.discard(17)
>>> a_set
{2, 3, 5, 7, 11, 13, 23}
```

- to discard a value that is not in the set has no effects;
- to remove a value that is not in the set raises a `KeyError` exception.

```
>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
>>> 30 in a_set
True
>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
>>> a_set.union(b_set)
{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}
>>> a_set.intersection(b_set)
{9, 2, 12, 5, 21}
>>> a_set.difference(b_set)
{195, 4, 76, 51, 30, 127}
>>> a_set.symmetric_difference(b_set)    '''(A ∪ B) \ (A ∩ B)'''
{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
>>>
>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}
>>> a_set.issubset(b_set)
True
>>> b_set.issuperset(a_set)
True
```

A dictionary is an unordered set of key-value pairs

- when you add a key to the dictionary you must also add a value for that key
- a value for a key can be changed at any time.

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...              1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> type(SUFFIXES)
<class 'dict'>
>>> SUFFIXES[1024]
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>> SUFFIXES
{1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
  1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> SUFFIXES[1000] = ['kilo', 'mega', 'giga', 'tera', 'peta', 'exa', 'zetta', 'yotta']
>>> SUFFIXES
{1000: ['kilo', 'mega', 'giga', 'tera', 'peta', 'exa', 'zetta', 'yotta'],
 1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

The syntax is similar to sets, but

- you list comma separate couples of key/value;
- {} is the empty dictionary

Note: you cannot have more than one entry with the same key.

Datatypes & Recursion

Walter Cazzola

Primitive Type
Boolean
Numbers
Collections
Lists
Tuples
Sets
Dictionaries
Strings
Bytes
Recursion
definition
iteration vs recursion
Hanoi's Towers
References

Python's strings are a sequence of unicode characters.

```
>>> s = 'The Russian for «Hello World» is «Привет мир»'
>>> s
'The Russian for «Hello World» is «Привет мир»'
>>> s[34]
'П'
>>> s+'!!!'
'The Russian for «Hello World» is «Привет мир»!!!'
>>> s[34:44]
'Привет мир'
```

Strings behave as lists: you can:

- get the string length with the **len** function;
- concatenate strings with the + operator;
- slicing works as well.

Note that ", ' and ''' (three-in-a-row quotes) can be used to define a string constant.

Python 3 supports formatting values into strings.

- that is, to insert a value into a string with a placeholder.

Looking back at the `humanize.py` example.

```
for suffix in SUFFIXES[multiple]:
   size /= multiple
   if size < multiple:
      return '{0:.1f} {1}'.format(size, suffix)
```

- `{0}`, `{1}`, ... are placeholders that are replaced by the arguments of `format()`;
- `:.1f` is a format specifier, it can be used to add space-padding, align strings, control decimal precision and convert number to hexadecimal as in C.

```
>>> '1000{0[0]} = 1{0[1]}'.format(humanize.SUFFIXES[1000])
1000KB = 1MB
```

Split multi-line strings on the carriage return symbol.

```
>>> s = '''To be, or not to be: that is the question:
... Whether 'tis nobler in the mind to suffer
... The slings and arrows of outrageous fortune,
... Or to take arms against a sea of troubles,
... And by opposing end them?'''
>>> s.split('\n')
['To be, or not to be: that is the question:',
 "Whether 'tis nobler in the mind to suffer",
 'The slings and arrows of outrageous fortune,',
 'Or to take arms against a sea of troubles,', 'And by opposing end them?']
```

To lowercase a sentence.

```
>>> print(s.lower())
to be, or not to be: that is the question:
whether 'tis nobler in the mind to suffer
the slings and arrows of outrageous fortune,
or to take arms against a sea of troubles,
and by opposing end them?
```

To count the occurrences of a string into another.

```
>>> print(s.lower().count('f'))
5
```

An immutable sequence of numbers (0-255) is a bytes object.

The byte literal syntax (b'') is used to define a bytes object.

- each byte within the byte literal can be an ASCII character or an encoded hexadecimal number from \x00 to \xff.

```
>>> by = b'abcd\x65'
>>> by += b'\xff'
>>> by
b'abcde\xff'
>>> len(by)
6
>>> by[5]
255
>>> by[0]=102
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

Bytes objects are immutable! Byte arrays can be changed.

```
>>> b_arr = bytearray(by)
>>> b_arr
bytearray(b'abcde\xff')
>>> b_arr[0]=102
>>> b_arr
bytearray(b'fbcde\xff')
```

# Recursion
## Definition: Recursive Function

A function is called recursive when it is defined through itself.

Example: Factorial.

- $5! = 5 * 4 * 3 * 2 * 1$
- Note that: $5! = 5 * 4!$, $4! = 4 * 3!$ and so on.

Potentially a recursive computation.

From the mathematical definition:

$$n! = \begin{cases} 1 & \text{if } n=0, \\ n*(n-1)! & \text{otherwise.} \end{cases}$$

When n=0 is the base of the recursive computation (axiom) whereas the second step is the inductive step.

Still, a function is recursive when its execution implies another invocation to itself.

- directly, i.e. in the function body there is an explicit call to itself;
- indirectly, i.e. the function calls another function that calls the function itself.

```python
def fact(n):
    return 1 if n<=1 else n*fact(n-1)

if __name__ == '__main__':
    for i in [5, 7, 15, 25, 30, 42, 100]:
       print('fact({0:3d}) :- {1}'.format(i, fact(i)))
```

```
[9:22]cazzola@hymir:~/esercizi-pa>python3 factorial.py
fact(  5) :- 120
fact(  7) :- 5040
fact( 15) :- 1307674368000
fact( 25) :- 15511210043330985984000000
fact( 30) :- 265252859812191058636308480000000
fact( 42) :- 1405006117752879898543142606244511569936384000000000
fact(100) :- 933262154439441526816992388562667004907159682643816214685929638952175999932
     299156089414639761565182862536979208272237582511852109168640000000000000000000000000
[9:22]cazzola@hymir:~/esercizi-pa>
```

# Recursion
## Execution: What's Happen?





```
[12:14]cazzola@hymir:~/esercizi-pa>python3
>>> import factorial
>>> factorial.fact(4)
24
```

```
def fact(n):
    return
        1
        if n<=1
        else n*fact(n-1)
```

It runs fact(4):

- a new frame with $n = 4$ is pushed on the stack;
- n is greater than 1;
- it calculates 4*fact(3)6, it returns 24

It runs fact(3):

- a new frame with $n = 3$ is pushed on the stack;
- n is greater than 1;
- it calculates 3*fact(2)2, it returns 6

It runs fact(2):

- a new frame with $n = 2$ is pushed on the stack;
- n is greater than 1;
- it calculates 2*fact(1)1, it returns 2

It runs fact(1):

- a new frame with $n = 1$ is pushed on the stack;
- n is equal to 1;
- it returns 1

# Recursion
## Side Notes on the Execution.





At any invocations the run-time environment creates an activation record or frame used to store the current values of:

- local variables, parameters and the location for the return value.

To have a frame for any invocation permits to:

- trace the execution flow;
- store the current state and restore it after the execution;
- avoid interferences on the local variables.

Warning:

Without any stopping rule, the inductive step will be applied "forever".

- Actually, the inductive step is applied until the memory reserved by the virtual machine is full.

Leonardo Pisano, known as Fibonacci, in 1202 in his book "Liber Abaci" faced the (quite unrealistic) problem of determining:

> "how many pairs of rabbits can be produced from a single pair if each pair begets a new pair each month and every new pair becomes productive from the second month on, supposing that no pair dies"

To introduce a sequence whose i-th member is the sum of the 2 previous elements in the sequence. The sequence will be soon known as the Fibonacci numbers.

# Recursion
## Case Study: Fibonacci Numbers (Cont'd)

Fibonacci numbers are recursively defined:

$$f(n) = \begin{cases} 0 & \text{if } n=0, \\ 1 & \text{if } n=1 \text{ or } n=2, \\ f(n-1)+f(n-2) & \text{otherwise.} \end{cases}$$

The implementation comes forth from the definition:

```
def fibo(n):
   return n if n<=1 else fibo(n-1)+fibo(n-2)

if __name__ == '__main__':
   for i in [5, 7, 15, 25, 30]:
      print('fibo({0:3d}) :- {1}'.format(i, fibo(i)))
```

```
[14:29]cazzola@hymir:~/esercizi-pa>python3 fibonacci.py
fibo(  5) :- 5
fibo(  7) :- 13
fibo( 15) :- 610
fibo( 25) :- 75025
fibo( 30) :- 832040
[14:30]cazzola@hymir:~/esercizi-pa>
```

# Recursion
## Recursion Easier & More Elegant

The recursive solution is more intuitive:

```
def fibo(n):
   return n if n<=1 else fibo(n-1)+fibo(n-2)
```

The iterative solution is more cryptic:

```
def fibo(n):
Fib1, Fib2, FibN = 0,1,1
if n<=1: return n
else:
   for i in range(2, n+1):
      FibN=Fib1+Fib2
      Fib1=Fib2
      Fib2=FibN
   return FibN
```

But ...

The iterative implementation is more efficient:

```
[16:20]cazzola@hymir:~/esercizi-pa>python3
>>> from timeit import Timer
>>> Timer('fibo(10)', 'from ifibonacci import fibo').timeit()
26.872473001480103
>>> Timer('fibo(10)', 'from fibonacci import fibo').timeit()
657.5257818698883
```

The overhead is mainly due to the creation of the frame but this also affects the occupied memory.
As an example, the call `fibo(1000)`

- gives an answer if calculated by the iterative implementation;
- raises a `RuntimeError` exception in the recursive solution.

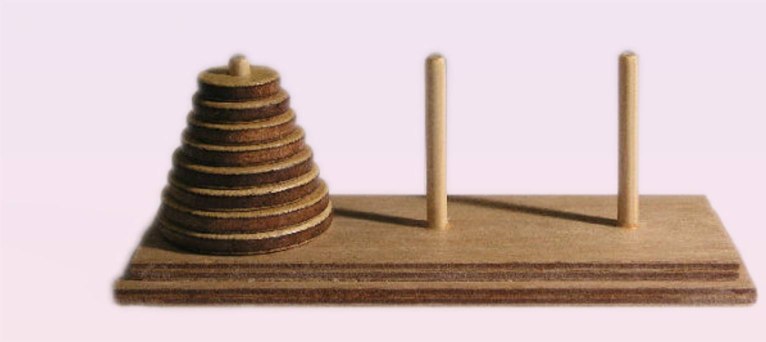
```
[16:45]cazzola@hymir:~/esercizi-pa>python3
>>> import ifibonacci
>>> import fibonacci
>>> ifibonacci.fibo(1000)
4346655768693745643568852767504062580256466051737178040248172908953655541794905189040387
9840079255169295922593080322634775209689623239873322471161642996440906533187938298969649
928516003704476137795166849228875
>>> fibonacci.fibo(1000)
     ...
File "fibonacci.py", line 2, in fibo
  return n if n<=1 else fibo(n-1)+fibo(n-2)
RuntimeError: maximum recursion depth exceeded in cmp
```

# The Towers of Hanoi
## Definition (Édouard Lucas, 1883)



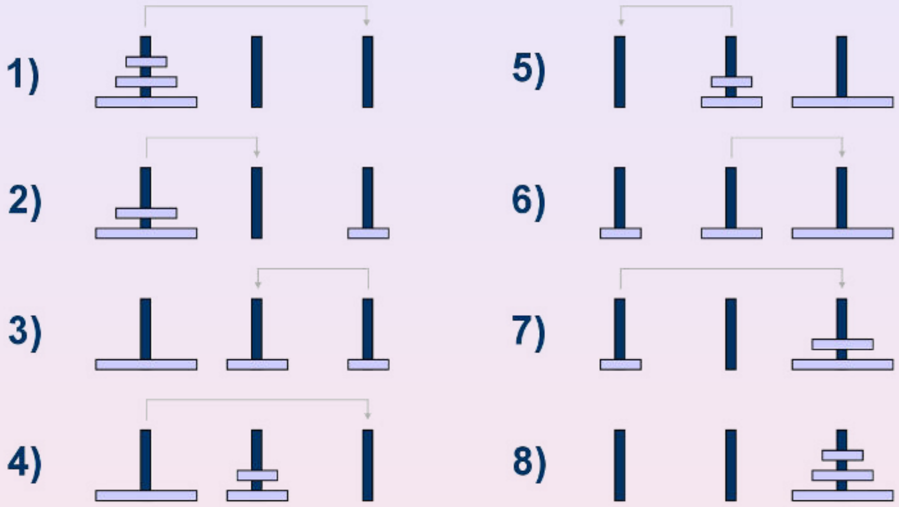
### Problem Description

There are 3 available pegs and several holed disks that should be stacked on the pegs. The diameter of the disks differs from disk to disk each disk can be stacked only on a larger disk.



The goal of the game is to move all the disks, one by one, from the first peg to the last one without ever violate the rules.

### 3-Disks Algorithm




### n-Disks Algorithm

**Base:** n=1, move the disk from the source (S) to the target (T);

**Step:** move n-1 disks from S to the first free peg (F), move the last disk to the target peg (T), finally move the n-1 disks from F to T.

# The Towers of Hanoi
## Python Implementation



```python
def display(pegs):
   for j in range(len(pegs[0])):
      for i in range(3):
         print('  {0}  '.format(pegs[i][j]), end="")
      print()
   print()

def move(pegs, source, target):
   s = pegs[source].count(0)
   t = pegs[target].count(0) - 1
   pegs[target][t] = pegs[source][s]
   pegs[source][s] = 0
   display(pegs)

def moveDisks(pegs, disks, source, target, free):
   if disks <= 1:
     print("moving from {0} to {1}".format(source, target))
     move(pegs, source, target)
   else:
     moveDisks(pegs, disks-1, source, free, target)
     print("moving from {0} to {1}".format(source, target));
     move(pegs, source, target);
     moveDisks(pegs, disks-1, free, target, source);

if __name__ == '__main__':
   pegs = [list(range(1,4)), [0]*3, [0]*3]
   print("Start!")
   display(pegs)
   moveDisks(pegs, 3, 0, 2, 1)
```

# The Towers of Hanoi
## 3-Disks Run

```
[0:12]cazzola@hymir:~/esercizi-pa>python3 hanoi.py
Start!                moving from 0 to 1    moving from 0 to 2    moving from 1 to 2
  1    0    0           0    0    0           0    0    0           0    0    0
  2    0    0           0    0    0           0    1    0           0    0    2
  3    0    0           3    2    1           0    2    3           1    0    3

moving from 0 to 2    moving from 2 to 1    moving from 1 to 0    moving from 0 to 2
  0    0    0           0    0    0           0    0    0           0    0    1
  2    0    0           0    1    0           0    0    0           0    0    2
  3    0    1           3    2    0           1    2    3           0    0    3
```

# The Towers of Hanoi







## The Myth

The myth tells about some Buddhist monks devout to Brahm■ should engage in solving the problem with 64 golden disks and when solved the world will end.

**Can we be quiet?**

How many operations will be necessary to end the computation?

At every call of `moveDisks()` (at least) two recursive calls to itself are done. This can be proved very close to $2^n$.

If we could move one disk per second we need:

$$2^{64} = 18\,446\,744\,073\,709\,551\,616 \text{ seconds}$$

that is about 586549 billions of years and the age of the whole universe is estimated of: 13.7 billions of years.

# References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Lutz.
  Learning Python.
  O'Reilly, third edition, November 2007.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.