

Basics of Python

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelles, 2025

Lists and Tuples

Tuples are like lists, but immutable. They can be written with round parentheses () or with commas only:

```
A = 1,2,3
```

```
A
```

```
A = 1, # Tuple with one element
```

```
A
```

Tuples can be used if one wants to return more than one value from a function, and they can be used for assigning multiple variables at once.

Exchanging two variables:

```
a,b = 1,2
```

```
a,b = b,a
```

GCD-algorithm:

```
def gcd( n1, n2 ) :  
    if n1 == 0 and n2 == 0 :  
        raise ArithmeticError( "gcd(0,0) does not exist" )  
    while n2 :  
        n1, n2 = n2, n1 % n2  
    return n1
```

n -th Fibonacci number:

```
def fib( n ) :  
    if n == 0 :  
        return 0  
  
    f1,f2 = 0,1  
    for i in range(1,n) :  
        f1, f2 = f2, f1 + f2  
    return f2
```

Strings

Strings can be formed with single or double quotes. There is no distinction. Strings that go over many lines can be formed with triple single or double quotes.

```
"this is a string"  
--> 'this is a string'
```

```
"""This is a long  
string"""  
'This is a long\nstring'
```

Strings can be concatenated with +, as in
"the rain " + "in spain".

Strings can be repeated by multiplication: Both 3*"xyz" and
"xyz"*3 result in 'xyzxyzxyz'

String Comparison

String comparison is implemented by usual comparison operators:

```
"abc" < "abcd"
```

```
"abd" > "abcd"
```

```
"xy" == "xy"
```

Format

string has a `format` method, which is useful for inserting substrings in a string. It works a bit like `printf`.

```
what,country,place = "rain", "Spain", "the plain"
```

```
"The {} in {} stays mainly in {}".format(  
    what, country, place )
```

```
what,country,place = "wind", "Kazakhstan", "Nur-Sultan"
```

```
"The {} in {} stays mainly in {}".format(  
    what, country, place )
```

It also works with numbers:

```
n = 5
```

```
"the {}-th Fibonacci number equals {}".format( n, fib(n))
```

Format f-Strings

In Python 3.6 or higher, you can use `f"` instead of `format()`:

```
what,country,place = "rain", "Spain", "the plain"
```

```
s = f"The {what} in {country} stays mainly in {place}"  
print(s)
```

```
a,b = 4,5  
print( f"{a} + {b} equals {a+b}" )
```

Splitting

Use `split()` to split a string into a list:

```
f = "the quick brown fox jumps over the lazy dog"  
f. split( )
```

You can split with an arbitrary substring:

```
f = "Astana Kazakhstan"  
f. split( "st" )  
f. split( "a" )
```

Note that Python has no `char` type. A character is just a string of length one.

Joining

The inverse operation of splitting is joining.

`s.join(lst)` constructs a new list by merging the elements in `lst` and putting `s` between them.

```
s = [ "aaa", "bbb", "ccc" ]
```

```
lst = "|".join(s)
```

```
lst # prints 'aaa|bbb|ccc'
```

```
lst = " && ".join(s)
```

```
list # prints 'aaa && bbb && ccc'
```

More About Lists

Lists can be indexed by []:

```
A = [1,2,3,4,5,6]
```

```
A[1]
```

```
A[-2] # Lists can be indexed from the end.
```

```
len(A) # Gets the length of a list.
```

A subsequence of a list is called **slice**. Slices are formed with a semicolon.

```
B = A[ 2 : 4 ]
```

```
# Select from 2 to 4 (not inclusive). The result
```

```
# is a completely new list.
```

```
B = A[ 2 : ] # List starts at A[2].
```

```
B = A[ : -2 ] # List ends at A[-2].
```

Strides

Slices can have a stride in them, a step distance:

```
A = [1,2,3,4,5,6,7,8,9]
```

```
B = A[ 1 : 6 : 2 ]
```

```
# Negative stride is also possible:
```

```
B = A [ 6 : 1 : -2 ]
```

```
# Pattern is [ start : end : step ]
```

```
# If step > 0, absent start means 0
```

```
# If step > 0, absent end means len( ).
```

```
# If step < 0, absent start means -1 (last element).
```

```
B = A [ :: 2 ] # These are two separate : :
```

```
B = A [ :: -1 ] # Reverses list
```

Strides (2)

Strides also work on strings and tuples:

```
"good morning"[::-1] # Reverses string.  
(1,2,3,4)[2:]      # Constructs a tuple.
```

Try :

```
lst = [1,2,3,4,5,6,7]  
l1,l2 = ( lst[0::2], lst[1::2] )
```

Strides (3)

Subranges can be used for replacing sublists, also for replacements that change the size:

```
L = [1,2,3,4,5,6]
```

```
L[2:4] = [] # Deletes
```

```
L[2:4] = [100,101,102,103,104] # Makes list longer.
```

It seems that **list** in Python is implemented in the same way as `std::vector`. Hence, it is not similar to `std::list` or Prolog lists. Appending and removing at the end is cheap, while removing and inserting from the middle or the beginning can be expensive.

Adding, Multiplying

Strings, lists and tuples can be concatenated with + and multiplied with *:

```
A = "good morning"
```

```
A + A
```

```
2 * A
```

```
A * 100
```

```
B = (1,2,3)
```

```
B + B
```

```
B * 20
```

Some More Useful Methods of List

```
lst. clear( )    # clears contents.
```

```
lst. index(x)    # Finds first occurrence of x,  
                # returns its index. Raises  
                # ValueError if not found.
```

```
list( it )      # Build list from iterator.  
                # (Note that all containers are  
                # automatically converted into an iterator.)
```

```
len( lst )      # Length of the list.
```

```
lst. append(n) # Appends an element to list.

lst. extend( it ) # Extend list from iterator

lst. pop( )      # Remove from end, returns
                 # removed element.

lst. reverse( ) # Reverse list in place.

del lst[i]      # Deletes i-th element,
                # same as # lst[i:i+1] = []
                # This is a method of lst.
                # (Writing lst.del(i) would be
                # more natural.)
```

Lists and Iterators

Everything that can be used in a `for` loop, is an **iterator**.

We already saw that lists are implicitly converted into iterators, when used with a `for` loop.

```
for i in lst :  
    print(i)
```

Similarly, iterators can be turned into lists with the `list()` function:

```
lst = list( range( 10, 20 ))
```

`range(10, 20)` is an iterator, because you can write:

```
for i in range( 10, 20 ) :  
    print(i)
```

Lists and Iterators (2)

If you want to enumerate the elements together with their indices, use

```
for i in enumerate( lst ) :  
    print(i)
```

`enumerate(lst)` takes an iterator and constructs a new iterator.

It can be used to construct another list:

```
lst = [1,2,3,4]  
lst = list( enumerate( list ) )  
lst # contains [(0, 1), (1, 2), (2, 3), (3, 4)]
```

Lists and Iterators (3)

Slice assignment works with iterators:

```
lst = [1,2,3,4]
lst[ 1 : 2 ] = range(10,20).
```

Note that almost everything can be implicitly converted into an iterator.

You can also reassign all elements in a list:

```
lst = [ 1,2,3,4 ]
lst[ : ] = range(10,20)
```

This is not the same as `lst = range(10,20)`. (Try it out!) We will learn more about iterators soon. They are fundamental to good Python programming.

Dictionaries

Dictionaries can be used for quick look up. Internally, they are implemented as **hash maps**. Python has no tree-based dictionaries.

Keys must be **hashable**. That means that they can be tested for equality, have a hash method, and are immutable.

```
d1 = { "one": "jeden", "two": "dwa", "three": "trzy",  
      "four": "cztery" }  
d2 = dict( one = "eins", two = "zwei", three = "drei",  
          four = "vier" )  
  
dict( it ) # Construct a dictionary from an iterator  
          # that yields pairs (as lists or as tuples).
```

Useful Functions

```
len( d )      # Number of entries in dictionary.
x in d        # True if x occurs as key in d.
x not in s    # True if x does not occur as key in d.

d[x]          # Returns value associate to x
del d[x]      # Remove the entry with key x.
d[x] =       # Assigns value to x.
              # The last two are methods of the
              # dictionary. (__delitem__, __setitem__)

d. pop(x)     # Remove the key from the dictionary,
              # and return the value that is
              # associated to it.

d. clear( )   # Clears the dictionary.
```

Obtaining Iterators from Dictionaries

Implicit conversion (to iterator) creates an iterator that enumerates only the keys. One can also create iterators explicitly:

```
d. keys( )      # Iterator of the keys.  
d. values( )   # Iterator of the values  
d. items( )    # Iterator of key/value pairs.  
  
enumerate(d)   # Enumerates pairs (i,key).
```

In the last expression, `d` is implicitly transformed into an iterator that enumerates the keys. After that, `enumerate()` changes this iterator into an iterator that adds counters to the enumerated keys.

Iteration Order

On older version of Python, iterator order was determined by the hash function (which means: unpredictable).

From Python 3.7 onward, iteration order is guaranteed to be the insertion order, older elements first.

The method `popitem()` also remembers order: It removes (and returns) the key/value pair whose key added last.

Sets

Python also has built-in **sets**. These are a bit like lists, but they don't store repeated elements:

```
s = { 1,2,3,1,2,3 }
```

```
s # prints {1,2,3}
```

```
s = set(it) # Constructs a set from an iterator.
```

```
s[i] # Does not work
```

```
len(s) # Number of elements (cardinality)
```

```
e = set( ) # emptyset, {} creates empty dictionary!
```

Sets (2)

```
x in s          # True if x occurs in s.
x not in s      # True if x does not occur in s

s.add(x)        # Add x to the set.
s.remove(x)     # Remove x from set, raise KeyError if
                # not present.
s.discard(x)    # Remove x from set if present.

s.pop( )        # Remove a random element from the set,
                # and return it, raise KeyError
                # if set is empty.
```

I suspect that set is implemented in the same way as dictionary. In all cases, its elements must be **hashable**.

Sets (3)

Some more set functions:

```
s. union( it )
    # Constructs union of it with the iterated values.
s. intersection( it )
    # Constructs intersection of s and the iterated
    # values.
s. difference( it )
    # Constructs s with the iterator values removed.
s. symmetric_difference( it )
    # Constructs s with iterator values removed, merged
    # with the iterator values that are not in s.

# These methods do not change s! They construct
# new sets.
```

Sets (4)

```
s. issubset( it ) # True if s1 is a subset of the  
# elements iterated by it.
```

Deque (1)

Despite their name, Python lists are implemented as `std::vector`. Therefore, they are not suitable as queue or dequeue.

If you need a long queue, then `import collections` and use `collections.deque`.

```
q. append(n)          # Append to the end of the queue
q. appendleft(n)     # Append to the front of the queue
```

```
q. pop( )            # Returns last element
                    # from queue, and returns it.
```

```
q. popleft( )        # Removes front of queue,
                    # and returns it.
```

Deque (2)

```
q. extend( it )
    # Add iterated elements to the end.

q. extendleft( it )
    # Add iterated elements to the front.

q[i]          # returns i-th element.
q[i:j]        # Does not work!
```

Frozenset

A `frozenset` is like a `set`, but it is immutable.

It can be constructed from an iterator, and after that, not modified any more.

This makes `frozenset` usable as dictionary key, or set members.

Frozenset supports

`union`, `intersection`, `difference`, `symmetric_difference`,
`issubset`, `in`.

It seems that there does not exist a frozen dictionary, even though that would be useful.

Sorting

In order to sort anything, use `sorted(it)`. It creates a sorted list of the elements that are generated by `it`.