[FIM] FONDAMENTI DI INFORMATICA
per medicina e chirurgia high tech

L01: Data Types

Dott. Giorgio De Magistris

demagistris@diag.uniroma1.it

Corso di Laurea in Medicina e Chirurgia High Tech

Sapienza
Università di Roma

I3S

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

DIAG

Dipartimento di Ingegneria Informatica, Automatica e Gestionale

Data

- Usually a program gets some data in input, processes the data and return some data in output
- In Python each data is represented as an object
- An object is just a sequence of bytes stored in memory
- Each object has a type, that tells me how to interpret that sequence of bytes

Mutable vs Immutable Types

- In Python there are Mutable and Immutable data types
- Mutable objects can be modified
- Immutable objects cannot be modified

```
>>> a = 1
>>> a += 1
>>> a
2
>>> a = (1,2)
>>> a[0]
1
>>> a[1]
2
>>> a[1] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Integer is an immutable type, here I'm not modifying the int object, i'm re-assigning the variable a

Variable

- An object is referenced through a variable
- The variable contains the address in memory of the referenced object
- An object can be referenced by multiple variables
- When an object is not referenced by any variable the memory is freed by the garbage collector

```
>>> a = 1
>>> hex(id(a))
'0x2014aa26930'
>>> b = 1
>>> hex(id(b))
'0x2014aa26930'
>>> c = "ciao"
>>> d = c
>>> hex(id(c))
'0x2014d25b2b0'
>>> hex(id(d))
'0x2014d25b2b0'
>>> c = None
>>> hex(id(c))
'0x7ffb3d3bacd8'
```

Assignment

- Only a variable can appear to the left side (lvalue) of an assignment A = B

- The member on the right side of the assignment is either a literal, an expression, a function call

```
>>> a = 1
>>> b = 2
>>> c = a + b + 3
>>> c
6
>>> c = abs(-5)
>>> c
5
>>>
```

Boolean

- There are two boolean objects:
  - True
  - False

- A boolean object is returned by a boolean operations or a comparison

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a = []
>>> b = []
>>> a is b
False
```

Boolean Operations

Operation	Result
`x or y`	if *x* is true, then *x*, else *y*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

Boolean Comparisons

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

NoneType

- The class NoneType has a single object (singleton): **None**
- It stands for “no value”
- It is assigned to variables that are not yet initialized
- It is returned by functions that do not return any value
- It supports no operation
- It always evaluate to False in a condition

```
>>> a = None
>>> type(a)
<class 'NoneType'>
>>> a == 0
False
>>> bool(a)
False
>>>
```

Numeric

- Numeric types are *int, float, complex*

- They are Immutable types

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

Collections

- Collections are objects that contain other objects, the most used collection types are:
  - list: mutable sequence of arbitrary objects
  - tuple: immutable sequence of arbitrary objects
  - range: immutable sequence of integers
  - string: immutable sequence of characters

  Sequences
  - set: unordered set of objects (mutable)
  - dictionary: set of <key,value> pairs, where key must be an hashable object (see later)

Sequences Operations

- These are the most common operations for sequences (**list, tuple, range, string**)

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Sequences Operations

- For mutable sequences (**list**) I have also operations that modify the object

```
1  # initialize an empty list
2  l = list()
3  # or equivalently
4  l = []
5  # initialize a list with
6  # some elements
7  l = [1,2,"hello"]
```

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from *s* (same as `del s[:]`)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop()` or `s.pop(i)`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*
`s.reverse()`	reverses the items of *s* in place

Sets

- Sets do not allow element indexing with the operator **[ ]** because there is no ordering between elements

- However they share some operations with sequences

- They define some new operations:
  - union(other)
  - difference(other)
  - intersection(other)
  - isdisjoint(other)
  - issubset(other)
  - issuperset(other)

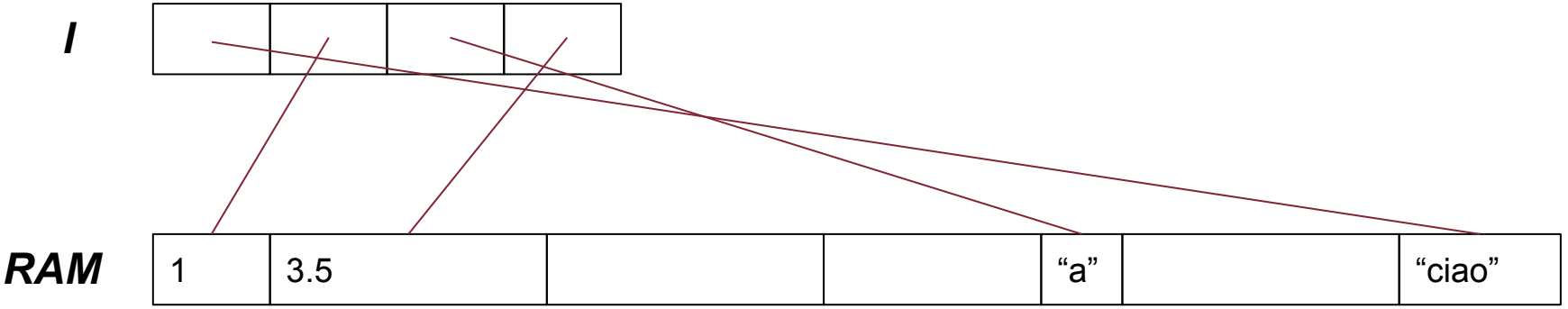
```
1  # initialize an empty set
2  s = set()
3  # or equivalently
4  s = {}
5  # initialize a set with
6  # some elements
7  s = {1,2,"hello"}
```

Operation
`x in s`
`x not in s`
`s + t`
`s * n` or `n * s`
~~`s[i]`~~
~~`s[i:j]`~~
~~`s[i:j:k]`~~
`len(s)`
`min(s)`
~~`max(s)`~~
~~`s.index(x[, i[, j]])`~~
`s.count(x)`

CPython List Implementation

- In CPython a list is implemented as an array of pointers:
  - The pointers are stored sequentially in memory
  - The objects are stored in arbitrary positions

l = [“ciao”, 1, “a”, 3.5]



Dictionary

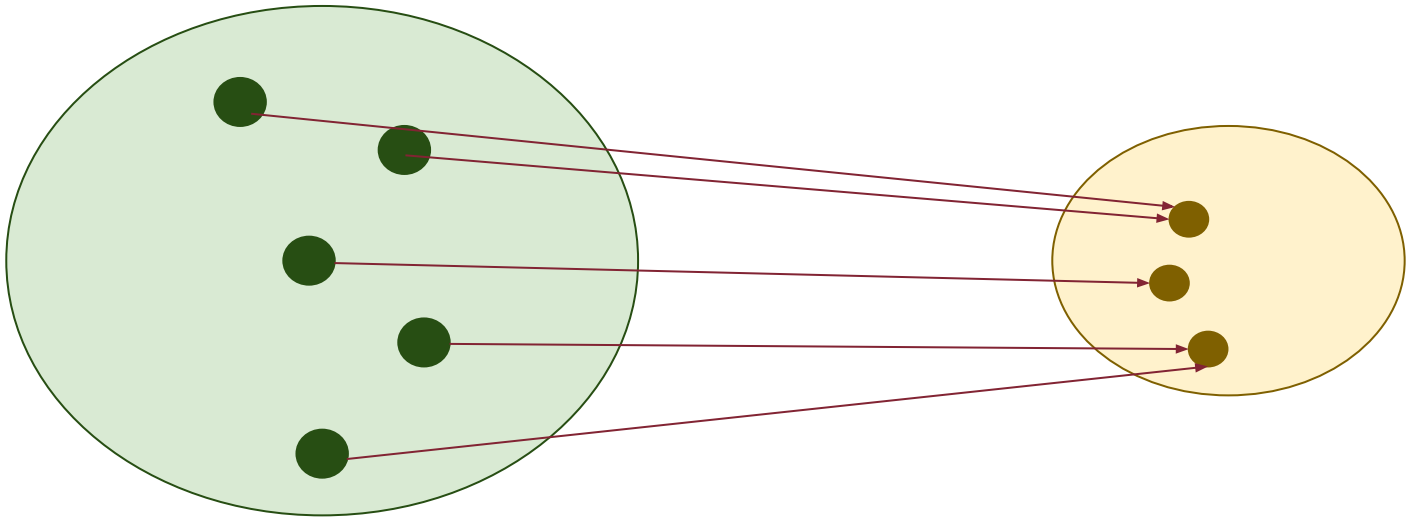
- A dictionary is a data structure that allows a fast access to the elements where elements are indexed by a unique key, it is implemented with an hash table.

Operation	Result
d = dict(one=1, two=2, three=3) d = {“one”:1, “two”:2, “three”:3}	initializes a dictionary with keys [“one”, “two”,”three”] and values [1,2,3]
list(d)	Return a list of all the keys used in the dictionary d
len(d)	Return the number of items in the dictionary d
d[key]	Return the item of d with key key. Raises a KeyError if key is not in the map.
d[key] = value	Set d[key] to value.
del d[key]	Remove d[key] from d. Raises a KeyError if key is not in the map.
key in d	Return True if d has a key key, else False

Hash Function

- An hash function is a “non-injective” function, i.e. a function that maps a large domain to a small codomain

- A good hash function should spread the elements of the domain uniformly across the codomain



Hash Table

- An hash table is a data structure that allows a quick access to <key,value> pairs elements
- The <key,value> pairs are stored in an array
- The index of the <key,value> pair is a function of the hash of the key
  - index = f(hash(key))
- If two two elements collide (have the same hash value, hence the same index) the second element is placed on the first free entry in the array
- In this way the cost of accessing one element (number of operations) is
  - evaluate hash function + scan sequentially until hit the desired key
  - 1 + avg number of collisions

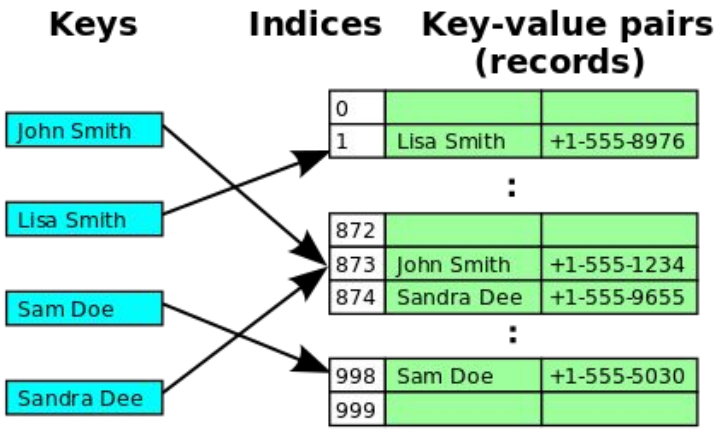


Image credit :
https://en.wikipedia.org/wiki/Linear_probing

string

- In Python text sequences are implemented by the str object
- a string (str object) is an immutable sequence of characters
- the methods that we have seen for immutable sequences apply also to strings
- a string is initialized putting text into single (‘) or double (“) quotes

```
>>> s1 = "hello"
>>> s2 = "world"
>>> s3 = s1 + s2
>>> s3
'helloworld'
>>> s3[1:-1:2]
'elwr'
>>> s3[-1::-1]
'dlrowolleh'
>>>
```

Methods and Attributes

- Each object has a list of attributes
- An attribute of the object is accessed using the following syntax
  - object.attribute_name
- A method is a callable attribute and it is called using the following syntax
  - *object.method_name(parameters)*
- The method receives as input the object itself and (optionally) a list of parameters
- It can either return a result or simply modify the object (if the object is mutable) and return nothing (None)
- dir(object) lists all the attributes of an object

```
>>> a = "hello"
>>> dir(a)
['__add__', '__class__', '__contains__', '__delat
tr__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__
', '__init_subclass__', '__iter__', '__le__', '__
len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__rep
r__', '__rmod__', '__rmul__', '__setattr__', '__s
izeof__', '__str__', '__subclasshook__', 'capital
ize', 'casefold', 'center', 'count', 'encode', 'e
ndswith', 'expandtabs', 'find', 'format', 'format
_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower'
, 'isnumeric', 'isprintable', 'isspace', 'istitle
', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
 'maketrans', 'partition', 'removeprefix', 'remov
esuffix', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'split
lines', 'startswith', 'strip', 'swapcase', 'title
', 'translate', 'upper', 'zfill']
```

Formatted string

- a formatted string or f-string is a string prefixed with the letter *f*
- it allows to insert other fields in a string, delimited by curly braces {}
- a formatted string is an expression that at run time is transformed to a constant literal by evaluating the expressions in the curly braces and inserting their values into the string

```
>>> a = 1
>>> b = 2
>>> s = f"a + b = {a} + {b} = {a+b}"
>>> s
'a + b = 1 + 2 = 3'
>>> type(s)
<class 'str'>
>>>
```

Formatting Options

- In formatted string i can also specify how the expression should be formatted
- To change the default behavior i need to specify a formatting option followed by a semicolon
    - f “formatted {expression:option} string”

```
>>> for i in range(11):
...     if i==0:
...             print(f"{'left':<10}{'center':^10}{'right':>10}")
...     else:
...             print(f"{i:<10}{i:^10}{i:>10}")
...
left        center       right
1             1             1
2             2             2
3             3             3
4             4             4
5             5             5
6             6             6
7             7             7
8             8             8
9             9             9
10           10            10
>>>
```

Alignment options

- Here are some alignment options

Option	Meaning
`'<'`	Forces the field to be left-aligned within the available space (this is the default for most objects).
`'>'`	Forces the field to be right-aligned within the available space (this is the default for numbers).
`'='`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default for numbers when '0' immediately precedes the field width.
`'^'`	Forces the field to be centered within the available space.

Numerical Presentation options

- Allow to specify how a numerical type should be formatted
- The presentation options for **integers** are

Type	Meaning
`'b'`	Binary format. Outputs the number in base 2.
`'c'`	Character. Converts the integer to the corresponding unicode character before printing.
`'d'`	Decimal Integer. Outputs the number in base 10.
`'o'`	Octal format. Outputs the number in base 8.
`'x'`	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
`'X'`	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9. In case `'#'` is specified, the prefix `'0x'` will be upper-cased to `'0X'` as well.
`'n'`	Number. This is the same as `'d'`, except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as `'d'`.

Numerical Presentation options

- The presentation options for **floats** are

Type	Meaning
`'e'`	Scientific notation. For a given precision `p`, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and `p` digits after the decimal point, for a total of `p + 1` significant digits. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and shows all coefficient digits for `Decimal`. If no digits follow the decimal point, the decimal point is also removed unless the `#` option is used.
`'E'`	Scientific notation. Same as `'e'` except it uses an upper case 'E' as the separator character.
`'f'`	Fixed-point notation. For a given precision `p`, formats the number as a decimal number with exactly `p` digits following the decimal point. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and uses a precision large enough to show all coefficient digits for `Decimal`. If no digits follow the decimal point, the decimal point is also removed unless the `#` option is used.
`'F'`	Fixed-point notation. Same as `'f'`, but converts `nan` to `NAN` and `inf` to `INF`.
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of `0` is treated as equivalent to a precision of `1`. The precise rules are as follows: suppose that the result formatted with presentation type `'e'` and precision `p-1` would have exponent `exp`. Then, if `m <= exp < p`, where `m` is -4 for floats and -6 for `Decimals`, the number is formatted with presentation type `'f'` and precision `p-1-exp`. Otherwise, the number is formatted with presentation type `'e'` and precision `p-1`. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the `'#'` option is used. With no precision given, uses a precision of `6` significant digits for `float`. For `Decimal`, the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than `1e-6` in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. Positive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision.
`'G'`	General format. Same as `'g'` except switches to `'E'` if the number gets too large. The representations of infinity and NaN are uppercased, too.
`'n'`	Number. This is the same as `'g'`, except that it uses the current locale setting to insert the appropriate number separator characters.
`'%'`	Percentage. Multiplies the number by 100 and displays in fixed (`'f'`) format, followed by a percent sign.
None	For `float` this is the same as `'g'`, except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point. The precision used is as large as needed to represent the given value faithfully. For `Decimal`, this is the same as either `'g'` or `'G'` depending on the value of `context.capitals` for the current decimal context. The overall effect is to match the output of `str()` as altered by the other format modifiers.