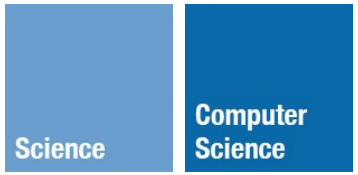
Introduction to Python



CS 331: Data Structures and Algorithms

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Agenda

- language overview
- built-in types, operators and constructs
- functions, classes, and modules
- stdlib: I/O, testing, timing

Not exhaustive!

- this is *not* a language course
  - Python is a means to an end (writing data structures & algorithms)
- *you* are responsible for mastering the language!

Quick Note on Tools

- More info and access to course server coming soon (accounts being set up)
- You can install Python 3.4 (download from python.org) or visit python.org/shell to follow along in class

IDE recommendation

- We will not be using an official IDE
  - Text editor (vim/emacs) + command line interpreter will suffice
- If you want/need one, I recommend PyCharm (from jetbrains.com)

§Overview

Python is …

interpreted,

dynamically-typed,

automatically memory-managed,

and supports *procedural*, *object-oriented*, *imperative* and *functional* paradigms

- Designed (mostly) by one man: Guido van Rossum (aka “benevolent dictator”)
- A single *reference implementation* (CPython)
- Current version (3) not backwards-compatible with previous (2), though latter is still widely used

PEP 20 -- The Zen of Python

```
    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!
```

i.e., for (small-ish) programming problems, there is usually an appropriate, idiomatic, *Pythonic* way of coding a solution

we'll review plenty of Pythonic examples in class that you can (and should!) emulate in your own work

§Types, Operators and Constructs (that matter to us)

Atomic types: `int`, `float`, `bool`

Compound types: `str`, `list`, `tuple`, `range`, `set`, `dict`

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

Boolean operators

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

Comparison operators

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*

Numeric operators

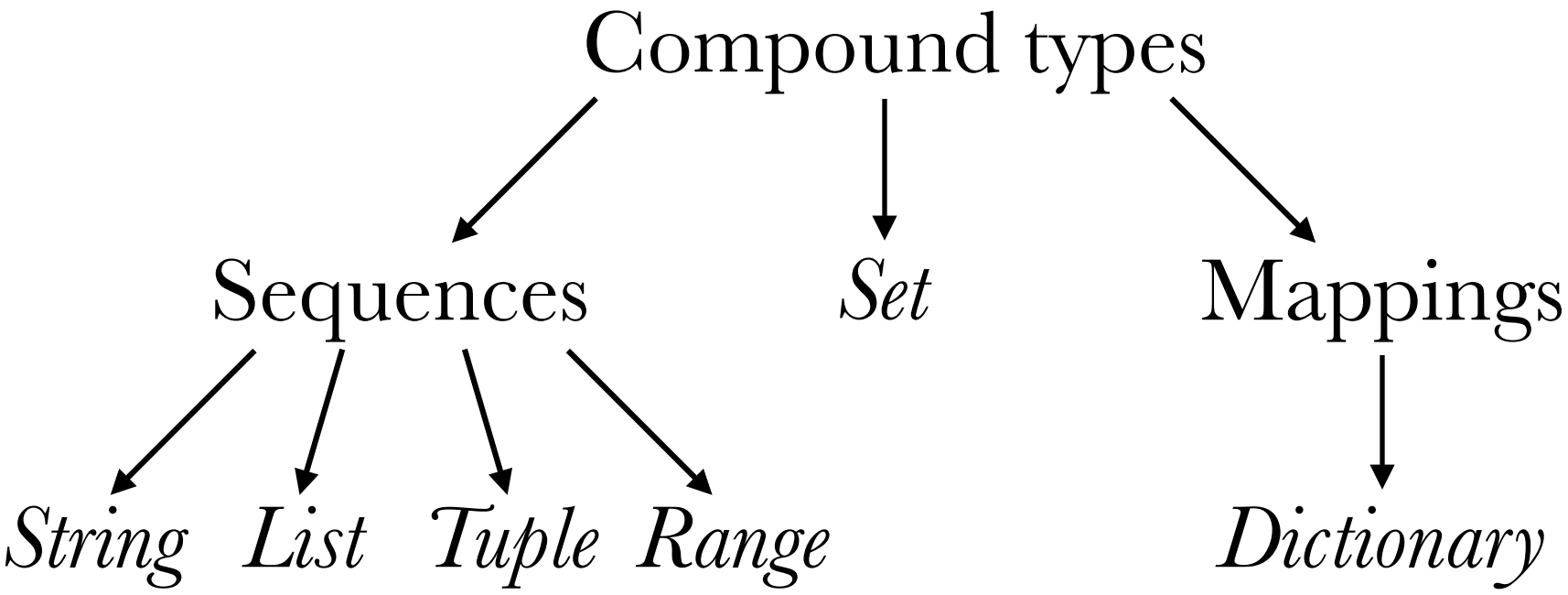
"Operators" either invoke *global functions* or *methods* on instances of built-in types

```
>>> 1 + 2
3
>>> (1).__add__(2)
3
```

Built-in (global) functions:

Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

https://docs.python.org/3/library/functions.html


Compound types
Sequences
Set
Mappings
String
List
Tuple
Range
Dictionary

Sequences are *ordered*, *indexable collections* of zero or more values.

Their contents may be:

- *homogeneous / heterogeneous*
- *immutable / mutable*

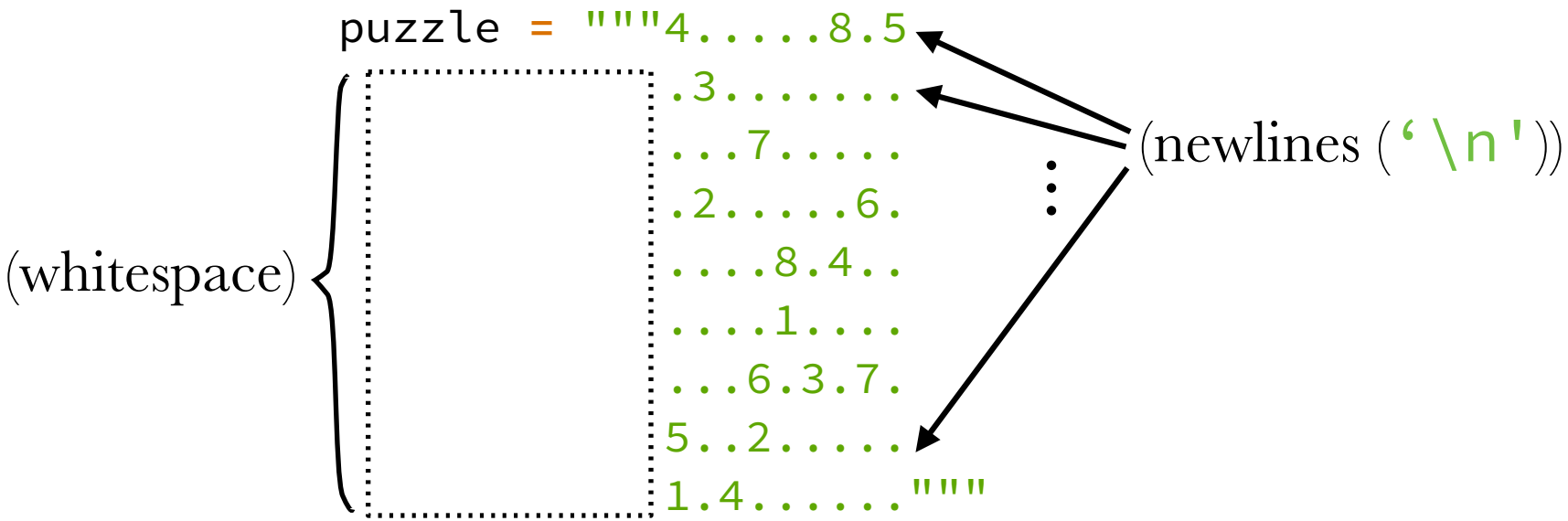
Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	*n* shallow copies of *s* concatenated
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *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*

Common sequence operations

String (`str` object): *immutable sequence* of characters; *string literals* are enclosed by single *or* double quotes

```
>>> 'hello'
'hello'
>>> 'hello' + ' ' + 'world'
'hello world'
>>> 'lo wo' in ('hello' + ' ' + 'world')
True
>>> 'world' * 3
'worldworldworld'
>>> len('hello')
5
>>> 'hello world'[3:10]
'lo worl’
>>> 'hello'[2:4] = 'ww'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Also, multi-line strings with three paired single/double quotes:



The almighty `list`: a *mutable, heterogeneous sequence*; list literals are comma-separated values enclosed by square brackets

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)`	extends *s* with the contents of *t* (same as `s[len(s):len(s)] = t`)
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`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] == x`
`s.reverse()`	reverses the items of *s* in place

Mutable sequence operations

```
>>> l = ['hello', 1, True, 3]
>>> l[2]
True
>>> 2 in l
False
>>> l.pop()
3
>>> del l[0]
>>> l
[1, True]
>>> l[0] = ['hello', 'there']
>>> l
[['hello', 'there'], True]
>>> l[-1] = 0
>>> l[1:-1] = [5, 4, 3, 2, 1]
>>> l
[['hello', 'there'], 5, 4, 3, 2, 1, 0]
```

```
puzzle = """4.....8.5
            .3.......
            ...7.....
            .2.....6.
            ....8.4..
            ....1....
            ...6.3.7.
            5..2.....
            1.4......"""

puzzle_rows = puzzle.split() # by default, splits on whitespace
```

```
[ '4.....8.5',
  '.3.......',
  '...7.....',
  '.2.....6.',
  '....8.4..',
  '....1....',
  '...6.3.7.',
  '5..2.....',
  '1.4......' ]
```

Iteration: `for` statement

- iterates over *iterable* types (e.g. sequences)

```
for c in "Hello rabbit":
    if c == 'l' or c == 'r':
        print('w', end='')
    elif c == ' ':
        print('-uh-', end='')
    else:
        print(c, end='')
```

```
Hewwo-uh-wabbit
```

```
for r in puzzle_rows:
    print(r, '->', r.count('.'), 'blanks')
```

```
4.....8.5 -> 6 blanks
.3.....  -> 8 blanks
...7.....  -> 8 blanks
.2.....6.  -> 7 blanks
....8.4..  -> 7 blanks
....1....  -> 8 blanks
...6.3.7.  -> 6 blanks
5..2.....  -> 7 blanks
1.4......  -> 7 blanks
```

```
blank_counts = []
for r in puzzle_rows:
    blank_counts.append(r.count('.'))
```

```
[6, 8, 8, 7, 7, 8, 6, 7, 7]
```

```
# more Pythonic: list-comprehension
blank_counts = [r.count('.') for r in puzzle_rows]
```

```
[6, 8, 8, 7, 7, 8, 6, 7, 7]
```

Tuples are immutable, heterogeneous sequences (comma-separated literals enclosed by parentheses)

Ranges are immutable sequences of numbers (created with `range` function, given start, stop, and optional step)

```
>>> (1.5, -2.0)
(1.5, -2.0)
>>> (5,) # a one-tuple requires the trailing comma
(5,)
>>> (5) # otherwise it's just a parenthesized value
5
>>> () # zero-element tuple
()
>>> 5 in range(0, 10)
True
>>> 10 in range(0, 10)
False
>>> 3 in range(0, 10, 2)
False
>>> list(range(20, 10, -3))
[20, 17, 14, 11]
>>> range(20, 10, -3)[2:]
range(14, 8, -3)
>>> list(range(14, 8, -3))
[14, 11]
```

recall, from Sudoku solution:

```
A1 A2 A3| A4 A5 A6| A7 A8 A9
B1 B2 B3| B4 B5 B6| B7 B8 B9
C1 C2 C3| C4 C5 C6| C7 C8 C9
---------+---------+---------
D1 D2 D3| D4 D5 D6| D7 D8 D9
E1 E2 E3| E4 E5 E6| E7 E8 E9
F1 F2 F3| F4 F5 F6| F7 F8 F9
---------+---------+---------
G1 G2 G3| G4 G5 G6| G7 G8 G9
H1 H2 H3| H4 H5 H6| H7 H8 H9
I1 I2 I3| I4 I5 I6| I7 I8 I9
```

```
rows = 'ABCDEFGHI'
cols = '123456789'

squares = []
for r in rows:
    for c in cols:
        squares.append(r + c)

print(len(squares))
print(squares[0:len(squares):5])
```

```
81
[ 'A1', 'A6', 'B2', 'B7', 'C3', 'C8', 'D4', 'D9',
  'E5', 'F1', 'F6', 'G2', 'G7', 'H3', 'H8', 'I4', 'I9']
```

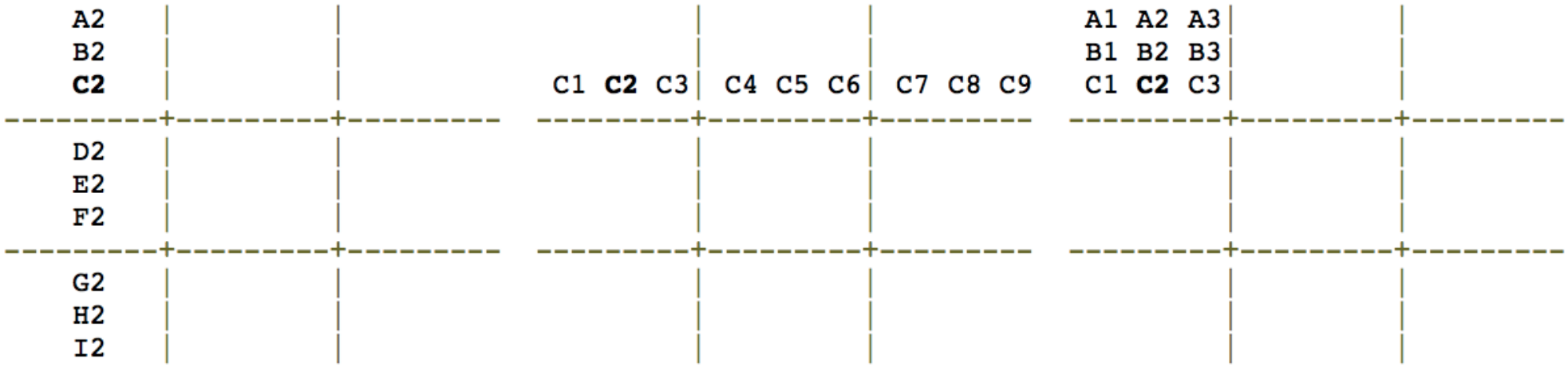
```
rows = 'ABCDEFGHI'
cols = '123456789'

# more Pythonic: list-comprehension
squares = [r+c for r in rows for c in cols]

print(len(squares))
print(squares[0:len(squares):5])
```

```
81
[ 'A1', 'A6', 'B2', 'B7', 'C3', 'C8', 'D4', 'D9',
  'E5', 'F1', 'F6', 'G2', 'G7', 'H3', 'H8', 'I4', 'I9']
```

identifying "units"



column unit *row unit* *box unit*

```
rows = 'ABCDEFGHI'
cols = '123456789'

c2_row = ['C'+c for c in cols]
c2_col = [r+'2' for r in rows]
c2_box = [r+c   for r in rows[0:3] for c in cols[0:3]]

c2_units = [c2_row, c2_col, c2_box]
```

```
[['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'],
 ['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2'],
 ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']]
```

```
rows = 'ABCDEFGHI'
cols = '123456789'

all_units = # ?

print(len(all_units))
print(all_units)
```

```
27
[['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9'],
 ...
 ['I1', 'I2', 'I3', 'I4', 'I5', 'I6', 'I7', 'I8', 'I9'],
 ['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1', 'I1'],
 ...
 ['A9', 'B9', 'C9', 'D9', 'E9', 'F9', 'G9', 'H9', 'I9'],
 ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3'],
 ...
 ['G7', 'G8', 'G9', 'H7', 'H8', 'H9', 'I7', 'I8', 'I9']]
```

hints:

- result is the *concatenation* of multiple list comprehension expressions
- each list comprehension must create a *list of lists* (each unit is a list)
- try to do it in pieces first!

```
 A1  A2  A3 |  A4  A5  A6 |  A7  A8  A9
 B1  B2  B3 |  B4  B5  B6 |  B7  B8  B9
 C1  C2  C3 |  C4  C5  C6 |  C7  C8  C9
------------+------------+------------
 D1  D2  D3 |  D4  D5  D6 |  D7  D8  D9
 E1  E2  E3 |  E4  E5  E6 |  E7  E8  E9
 F1  F2  F3 |  F4  F5  F6 |  F7  F8  F9
------------+------------+------------
 G1  G2  G3 |  G4  G5  G6 |  G7  G8  G9
 H1  H2  H3 |  H4  H5  H6 |  H7  H8  H9
 I1  I2  I3 |  I4  I5  I6 |  I7  I8  I9
```

```
row_units = [[r+c for c in cols] for r in rows]
```

```
[['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9'],
 ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8', 'B9'],
 ['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'],
 ...
 ['I1', 'I2', 'I3', 'I4', 'I5', 'I6', 'I7', 'I8', 'I9']]
```

```
col_units = [[r+c for r in rows] for c in cols]
```

```
[['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1', 'I1'],
 ['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2'],
 ['A3', 'B3', 'C3', 'D3', 'E3', 'F3', 'G3', 'H3', 'I3'],
 ['A9', 'B9', 'C9', 'D9', 'E9', 'F9', 'G9', 'H9', 'I9']]
```

“northwest” box unit:

```
['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
```

```
nw_box_unit = [r+c for r in rows[0:3] for c in cols[0:3]]
```

```
box_units = [[r+c for r in rs for c in cs]
             for rs in ?
             for cs in ?]
```

```
box_units = [[r+c for r in rs for c in cs]
             for rs in (rows[0:3], rows[3:6], rows[6:9])
             for cs in (cols[0:3], cols[3:6], cols[6:9])]
```

```
[['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3'],
 ['A4', 'A5', 'A6', 'B4', 'B5', 'B6', 'C4', 'C5', 'C6'],
 ['A7', 'A8', 'A9', 'B7', 'B8', 'B9', 'C7', 'C8', 'C9'],
 ...
 ['G4', 'G5', 'G6', 'H4', 'H5', 'H6', 'I4', 'I5', 'I6'],
 ['G7', 'G8', 'G9', 'H7', 'H8', 'H9', 'I7', 'I8', 'I9']]
```

```
box_units = [[r+c for r in rs for c in cs]
             for rs in ('ABC', 'DEF', 'GHI')
             for cs in ('123', '456', '789')]
```

```
[['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3'],
 ['A4', 'A5', 'A6', 'B4', 'B5', 'B6', 'C4', 'C5', 'C6'],
 ['A7', 'A8', 'A9', 'B7', 'B8', 'B9', 'C7', 'C8', 'C9'],
 ...
 ['G4', 'G5', 'G6', 'H4', 'H5', 'H6', 'I4', 'I5', 'I6'],
 ['G7', 'G8', 'G9', 'H7', 'H8', 'H9', 'I7', 'I8', 'I9']]
```

```
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])
```

```
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])

c2_units = # ?
```

```
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])

c2_units = [u for u in all_units if 'C2' in u]
            # filtered list comprehension
```

```
[['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'],
 ['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2'],
 ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']]
```

```
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])

square_units = [(s, [u for u in all_units if s in u])
                for s in squares]

print(square_units[0])
```

```
('A1',
 [['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9'],
  ['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1', 'I1'],
  ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']])
```

```
to_find = 'C2'
for entry in square_units:
    if entry[0] == to_find:
        units = entry[1]
        break
else: # enter when loop exhausts all values (i.e., no break)
    units = None # special value that represents ... nothing

print(units)
```

```
[['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'],
 ['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2'],
 ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']]
```

```
for entry in square_units:
    if entry[0] == to_find:
        units = entry[1]
        break
else: # enter when loop exhausts all values (i.e., no break)
    units = None # special value that represents ... nothing
```

- Not very efficient!
- We need to “look up” the units/peers of a given square very frequently while solving a puzzle

Enter *mapping* compound type: `dict`

Mutable structure that (efficiently) maps *keys* to arbitrary *values*.

- keys must be *hashable* — more on this later, but for now assume this includes all *immutable* built-in types

Operation	Result
`len(d)`	returns the number of mappings in dictionary `d`
`d[k]`	returns the value in dictionary `d` with key `k`
`d[k] = val`	sets the value for key `k` in dictionary `d` to `val`
`del d[k]`	removes the mapping for key `k`
`k in d`	returns `True` if `d` has key `k`, else `False`
`k not in d`	equivalent to `not key in d`
`d.items()`	returns an iterator over `d`'s (key, value) pairs
`d.keys()`	returns an iterator over `d`'s keys
`d.values()`	returns an iterator over `d`'s values

Dictionary operations

```
>>> alter_egos = {'Superman': 'Clark Kent',
                  'Batman': 'Bruce Wayne',
                  'Spiderman': 'Peter Parker'}
>>> alter_egos['Superman']
'Clark Kent'
>>> alter_egos['Iron Man'] = 'Tony Stark'
>>> alter_egos
{'Batman': 'Bruce Wayne',
 'Iron Man': 'Tony Stark',
 'Superman': 'Clark Kent',
 'Spiderman': 'Peter Parker'}
>>> alter_egos['Dr. Doom']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Dr. Doom'
```

```
# aside: exception handling

alter_egos = {'Superman': 'Clark Kent',
              'Batman': 'Bruce Wayne',
              'Spiderman': 'Peter ParkerÕ}

try:
    dd_ego = alter_egos['Dr. Doom']
except:
    print("Dr. Doom's alter ego not known")
else:
    print('Dr. Doom =', dd_ego)
finally:
    print('Thanks for playing!')
```

```
Dr. Doom's alter ego not known
Thanks for playing!
```

```
# alternative: use get method with a default

print(alter_egos.get('Superman', 'unknown'))
print(alter_egos.get('Dr. Doom', 'unknown'))
```

```
Clark Kent
unknown
```

```
# if no default, get returns None if no mapping

print(alter_egos.get('Superman'))
print(alter_egos.get('Dr. Doom'))
```

```
Clark Kent
None
```

```
# note: keys are not stored or returned in order!

alter_egos = {'Superman': 'Clark Kent',
              'Batman': 'Bruce Wayne',
              'Spiderman': 'Peter Parker',
              'Iron Man': 'Tony Stark'}

for k, v in alter_egos.items():
    print(k, '=>', v)
```

```
Iron Man => Tony Stark
Superman => Clark Kent
Batman => Bruce Wayne
Spiderman => Peter Parker
```

```
# use sorted() to explicitly sort keys

alter_egos = {'Superman': 'Clark Kent',
              'Batman': 'Bruce Wayne',
              'Spiderman': 'Peter Parker',
              'Iron Man': 'Tony Stark'}

for superhero in sorted(alter_egos.keys()):
    print(superhero, '=>', alter_egos[superhero])
```

```
Batman => Bruce Wayne
Iron Man => Tony Stark
Spiderman => Peter Parker
Superman => Clark Kent
```

```
# e.g., constructing a dictionary

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
alpha_num = {}

for i in range(0, len(alpha)):
    alpha_num[alpha[i]] = i+1 # create new dict entry
```

```
{'X': 24, 'M': 13, 'C': 3, 'S': 19, 'B': 2,
 'V': 22, 'F': 6, 'A': 1, 'U': 21, 'R': 18,
 'Z': 26, 'D': 4, 'P': 16, 'I': 9, 'Q': 17,
 'T': 20, 'N': 14, 'G': 7, 'W': 23, 'O': 15,
 'H': 8, 'L': 12, 'K': 11, 'Y': 25, 'J': 10,
 'E': 5}
```

```
# Pythonically: dict comprehension

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
alpha_num = {alpha[i]: i+1 for i in range(0, len(alpha))}
```

```
{'X': 24, 'M': 13, 'C': 3, 'S': 19, 'B': 2,
 'V': 22, 'F': 6, 'A': 1, 'U': 21, 'R': 18,
 'Z': 26, 'D': 4, 'P': 16, 'I': 9, 'Q': 17,
 'T': 20, 'N': 14, 'G': 7, 'W': 23, 'O': 15,
 'H': 8, 'L': 12, 'K': 11, 'Y': 25, 'J': 10,
 'E': 5}
```

back to Sudoku:

```
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])

square_units = [(s, [u for u in all_units if s in u])
                for s in squares]

for entry in square_units:
    if entry[0] == 'C2':
        c2_units = entry[1]
        break
```

```
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])

# use a dictionary for mapping
units = {s: [u for u in all_units if s in u]
         for s in squares}

c2_units = units['C2']
```

```
units = {s: [u for u in all_units if s in u]
         for s in squares}

c2_units = units['C2']

c2_peers = [sq for u in c2_units for sq in u]

print(len(c2_peers))
print(c2_peers)
```

```
27
['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9',
 'A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2',
 'A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
```

```
27
['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9',
 'A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2',
 'A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
```

Problems:

1. List contains duplicate peers
2. List of peers of a square should not include the square itself! (C2, above)

Enter (last) compound type: `set`

- *unordered* collection with *no duplicates*
- elements must also be hashable
- create from sequences with `set()`
- set literals with `{...}`

Operation	Result
`len(s)`	returns the number of items in set `s`
`x in s`	tests if `x` is in set `k`
`x not in s`	equivalent to `not x in s`
`s1.isdisjoint(s2)`	returns `True` if set `s1` has no elements in common with set `s2`
`s1 <= s2`	tests if every element of set `s1` is in set `s2`
`s1 | s2`	returns a new set with elements from sets `s1` and `s2`
`s1 & s2`	returns a new set with elements common to sets `s1` and `s2`
`s1 - s2`	returns a new set with elements from sets `s1` not in `s2`
`s1 ^ s2`	returns a new set with elements from either `s1` or `s2` but not both

Set operations

```
units = {s: [u for u in all_units if s in u]
         for s in squares}

c2_units = units['C2']

c2_peers = [sq for u in c2_units for sq in u]

print(len(c2_peers))
print(c2_peers)
```

```
27
['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9',
 'A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2',
 'A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
```

```
units = {s: [u for u in all_units if s in u]
         for s in squares}

c2_units = units['C2']

c2_peers = set([sq for u in c2_units for sq in u]) - {'C2'}

print(len(c2_peers))
print(sorted(c2_peers)) # sorting into list for inspection
```

```
20
['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C3', 'C4',
 'C5', 'C6', 'C7', 'C8', 'C9', 'D2', 'E2', 'F2', 'G2',
 'H2', 'I2']
```

```
units = {s: [u for u in all_units if s in u]
         for s in squares}

peers = {s: (set([sq for u in units[s] for sq in u]) - {s})
         for s in squares }

c2_peers = peers['C2']

print(len(c2_peers))
print(sorted(c2_peers))
```

```
20
['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C3', 'C4',
 'C5', 'C6', 'C7', 'C8', 'C9', 'D2', 'E2', 'F2', 'G2',
 'H2', 'I2']
```

```
rows = 'ABCDEFGHI'
cols = '123456789'

# list of square names
squares = [r+c for r in rows for c in cols]

# list of lists of squares (each sublist is a unit)
all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])

# dictionary mapping a square to a list of lists of squares
units = {s: [u for u in all_units if s in u]
         for s in squares}

# dictionary mapping a square to a set of squares
peers = {s: (set([sq for u in units[s] for sq in u]) - {s})
         for s in squares }
```

```
squares = [r+c for r in rows for c in cols]
assert(len(squares) == 81)

all_units = ([[r+c for c in cols] for r in rows] +
             [[r+c for r in rows] for c in cols] +
             [[r+c for r in rs for c in cs]
              for rs in ('ABC', 'DEF', 'GHI')
              for cs in ('123', '456', '789')])
assert(len(all_units) == 27)

units = {s: [u for u in all_units if s in u] for s in squares}
assert(all(len(units[s]) == 3 for s in squares))

peers = {s: (set([sq for u in units[s] for sq in u]) - {s})
         for s in squares }
assert(all(len(peers[s]) == 20 for s in squares))
```

Built-in “all” and “any” functions:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True


def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

§Functions

```
def foo():
    pass # special statement -- does nothing!
         # (useful as a placeholder)

def bar():
    """Calls foo. This is a function *docstring*.
    It should briefly describe what the function
    does. I won't always use them in slides, but
    you should!"""
    foo() # call foo
```

```
def mysum(x, y):
    return x+y # works for all plus-able things!

print(mysum(1, 2))
print(mysum('hello', 'world'))
```

```
3
helloworld
```

```
def mysum_v2(vals):
    """This version takes a list of vals to add"""
    accum = 0 # ← restricts to numbers
    for item in vals:
        accum += item
    return accum

print(mysum_v2([1, 2, 3, 4, 5]))
print(mysum_v2(range(10)))
print(mysum_v2(['hello', 'world']))
```

```
15
45
Traceback (most recent call last):
  File "functions.py", line 12, in <module>
    print(mysum_v2(['hello', 'world']))
  File "functions.py", line 7, in mysum_v2
    accum += item
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

```
def mysum_v3(vals, start):
    """This version also takes a start value"""
    accum = start
    for item in vals:
        accum += item
    return accum

print(mysum_v3([1, 2, 3, 4, 5], 0))
print(mysum_v3(['hello', 'world'], ''))
print(mysum_v3([(1, 2), (), (5,)], ()))
```

```
15
helloworld
(1, 2, 5)
```

```
def mysum_v4(vals, start=0):
    """This version defaults to using 0 for start"""
    accum = start
    for item in vals:
        accum += item
    return accum

print(mysum_v4([1, 2, 3, 4, 5]))
print(mysum_v4(['hello', 'world'], ''))
print(mysum_v4(['hello', 'world'], start=''))
print(mysum_v4(start='-->', vals=['a', 'b', 'c']))
```

```
15
helloworld
helloworld
-->abc
```

```
def mysum_v5(*vals, start=0):
    """This version takes an arbitrary number of
    arguments that are automatically bundled into
    the vals variable."""
    accum = start
    for item in vals:
        accum += item
    return accum

print(mysum_v5(1, 2, 3, 4))
print(mysum_v5('hello', ' ', 'world', start='>'))
```

```
10
>hello world
```

```
def mysum_v5(*vals, start=0):
    """This version takes an arbitrary number of
    arguments that are automatically bundled into
    the vals variable."""
    accum = start
    for item in vals:
        accum += item
    return accum

print(mysum_v5(start='>', 'foo', 'bar'))
```

```
File "functions.py", line 49
    print(mysum_v5(start='>', 'foo', 'bar'))
                                ^
SyntaxError: non-keyword arg after keyword arg
```

```
def mysum_v5(*vals, start=0):
    """This version takes an arbitrary number of
    arguments that are automatically bundled into
    the vals variable."""
    accum = start
    for item in vals:
        accum += item
    return accum

args = [10, 20, 30] + list(range(40, 110, 10))
print(mysum_v5(*args)) # "unpack" args from list
```

```
550
```

```
def reduce(combiner, *vals, start=0):
    """Combines all items in vals with the provided
    combiner function and start value"""
    accum = start
    for item in vals:
        accum = combiner(accum, item)
    return accum

def add(m, n):
    return m+n

print(reduce(add, 1, 2, 3, 4))
print(reduce(add, 'hello', 'world', start=''))
```

```
10
helloworld
```

```
def reduce(combiner, *vals, start=0):
    """Combines all items in vals with the provided
    combiner function and start value"""
    accum = start
    for item in vals:
        accum = combiner(accum, item)
    return accum

def mult(m, n):
    return m*n

print(reduce(mult, 1, 2, 3, 4))
print(reduce(mult, 1, 2, 3, 4, start=1))
```

```
0
24
```

```
def add(m, n):
    return m+n

def mult(m, n):
    return m*n
```

... a bit verbose for functions defined solely to be passed as arguments

```
# "lambda" defines single-expression functions
add  = lambda m, n: m+n
mult = lambda m, n: m*n
pow_of_2 = lambda x: 2**x

add(5, 6)     # => 11
mult(5, 6)    # => 30
pow_of_2(10) #=> 1024

# "anonymous" lambda application
(lambda x, y: x**y)(2, 10) #=> 1024
```

```
def reduce(combiner, *vals, start=0):
    accum = start
     for item in vals:
         accum = combiner(accum, item)
     return accum

print(reduce(lambda x,y: x*y, 1, 2, 3, 4, start=1))
print(reduce(lambda sos,n: sos + n**2, 1, 2, 3, 4))
print(reduce(lambda total, s: total + len(s),
             'hello', 'beautiful', 'world'))
print(reduce(lambda s, l: s & set(l), # set intersect
             range(0,10), range(5,20), range(8,12),
             start=set(range(0,100))))
```

```
24
30
19
{8, 9}
```

```
// sorting strings by length in Java
String[] names = {"ingesting", "cakes", "is", "fun"};
Arrays.sort(names, new Comparator<String>() {
    // custom Comparator object needed
    public int compare(String s, String t) {
      return s.length() - t.length();
    }
  });
for (String n: names) {
  System.out.println(n);
}
```

```
is
fun
cakes
ingesting
```

Python global function “sorted”:

`sorted`(iterable[, key][, reverse])

Return a new sorted list from the items in iterable.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

```
>>> sorted([-3, 5, -1, 0])
[-3, -1, 0, 5]
>>> sorted([-3, 5, -1, 0], key=abs)
[0, -1, -3, 5]
>>> sorted(['ingesting', 'cakes', 'is', 'fun'])
['cakes', 'fun', 'ingesting', 'is']
>>> sorted(['ingesting', 'cakes', 'is', 'fun'],
           key=lambda s: len(s))
['is', 'fun', 'cakes', 'ingesting']
>>> sorted(['ingesting', 'cakes', 'is', 'fun'],
           key=len)
['is', 'fun', 'cakes', 'ingesting']
>>> sorted(['ingesting', 'cakes', 'is', 'fun'],
           key=lambda s: s.count('i'),
           reverse=True)
['ingesting', 'is', 'cakes', 'fun']
```

```
def print_char_sheet(name,          # normal arg
                     *inventory,    # arbitrary num of args in tuple
                     race='Human', # keyword arg with default
                     **info):       # arbitrary keyword args in dict
    """Demonstrates all sorts of arg types."""
    print('Name: ', name)
    print('Race: ', race)
    print('Inventory:')
    for item in inventory:
        print(' -', item)
    for k in sorted(info.keys()):
        print('*', k, '=', info[k])



print_char_sheet('Joe', 'scissors', 'phone',
                 home='Chicago')

print_char_sheet('Mary', 'sword', race='Elf')

print_char_sheet('Brad', 'axe', 'torch', 'match',
                 status='single', race='Dwarf',
                 height='short')
```

```
Name:  Joe
Race:  Human
Inventory:
 - scissors
 - phone
* home = Chicago

Name:  Mary
Race:  Elf
Inventory:
 - sword

Name:  Brad
Race:  Dwarf
Inventory:
 - axe
 - torch
 - match
* height = short
* status = single
```

```
def i_take_3_args(foo, bar, baz):
    print('Got', foo, bar, baz)

i_take_3_args(1, 2, 3)                  # positional args

i_take_3_args(baz=3, foo=1, bar=2)      # named args

args = {'bar': 2, 'baz': 3, 'foo': 1}
i_take_3_args(**args)                   # args from dictionary
```

```
Got 1 2 3
Got 1 2 3
Got 1 2 3
```

Digression: on assignment (=) statements

`target = expression`

1. if target is a name, evaluate expression and bind target to resulting object
2. if target is subscripted or sliced, follow (mutable) target's assignment rule
3. if target and expression are *lists*, evaluate all expressions then assign corresponding items

```
>>> a = 10
>>> b = a + 10
>>> tmp = a
>>> a = b
>>> b = tmp
>>> (a, b)
(20, 10)
>>> l = [10, 9, 8, 7, 6, '...']
>>> l[-1] = 5
>>> l[len(l):] = [4, 3, 2, 1, 'Bang!']
>>> l
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'Bang!']
```

```
>>> a, b, c = (2**0, 2**1, 2**2)
>>> (a, b, c)
(1, 2, 4)
>>> person = ['John', 'Doe']
>>> first, last = person
>>> first, last
('John', 'Doe')
>>> animal, *etc = 'Lions', 'Tigers', 'Bears', 'Oh my'
>>> [animal, etc]
['Lions', ['Tigers', 'Bears', 'Oh my']]
>>> _, *ns = range(0, 100, 9)
>>> ns
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99]
>>> a, b = b, a
>>> c, d, e = c+1, c+2, c+3
>>> [a, b, c, d, e]
[2, 1, 5, 6, 7]
```

```
def fibonacci(nth):
    a, b = 1, 1
    for _ in range(nth-1):
        a, b = b, a+b
    return a

print([fibonacci(i) for i in range(1, 15)])
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Control flow recap:

- assignment (single/multiple): `=`
- branching: `if...elif...else`
- looping: `for`, `while`;
  `break`, `continue`, `else`
- exceptions: `try...except...`
  `else...finally`
- doing nothing: `pass`

```
puzzle = """4.....8.5
            .3.......
            ...7.....
            .2.....6.
            ....8.4..
            ....1....
            ...6.3.7.
            5..2.....
            1.4......"""


def parse_puzzle(puz):
    puzzle = [c if c.isdigit() else None
              for c in puz if c not in ' \n']
    assert(len(puzzle) == 81)
    return {squares[i]: puzzle[i]
            for i in range(0, len(squares))}

print(parse_puzzle(puzzle))
```

```
{'A1': '4',
 'A2': None,
 'A3': None,
 'A4': None,
 'A5': None,
 'A6': None,
 'A7': '8',
 'A8': None,
 'A9': '5',
 ...
 'I1': '1',
 'I2': None,
 'I3': '4',
 'I4': None,
 'I5': None,
 'I6': None,
 'I7': None,
 'I8': None,
 'I9': None}
```

```
sol = {s: '123456789' for s in squares}

for sq, val in parse_puzzle(puzzle).items():
    if val:
        assign(sol, sq, val)
```

```
def assign(sol, sq, val):
    # eliminate all values from square sq except val


def eliminate(sol, sq, val):
    # remove value val from square sq, then
    # (1) if there's only one value left, eliminate
    #     it from all my peers
    # (2) if we just eliminated the second-to-last
    #     entry for a given value in some unit,
    #     assign the value to the final square
```

```
def assign(sol, sq, val):
    for other in sol[sq].replace(val, ''):
        eliminate(sol, sq, other)

def eliminate(sol, sq, val):
    # remove value val from square sq, then
    # (1) if there's only one value left, eliminate
    #     it from all my peers
    # (2) if we just eliminated the second-to-last
    #     entry for a given value in some unit,
    #     assign the value to the final square
```

```
def assign(sol, sq, val):
    for other in sol[sq].replace(val, ''):
        eliminate(sol, sq, other)

def eliminate(sol, sq, val):
    sol[sq] = sol[sq].replace(val, '')
    # (1) if there's only one value left, eliminate
    #     it from all my peers
    # (2) if we just eliminated the second-to-last
    #     entry for a given value in some unit,
    #     assign the value to the final square
```

```
def assign(sol, sq, val):
    for other in sol[sq].replace(val, ''):
        eliminate(sol, sq, other)

def eliminate(sol, sq, val):
    sol[sq] = sol[sq].replace(val, '')
    if len(sol[sq]) == 1:
        last = sol[sq][0]
        for p in peers[sq]:
            eliminate(sol, p, last)
    # (2) if we just eliminated the second-to-last
    #     entry for a given value in some unit,
    #     assign the value to the final square
```

```
def assign(sol, sq, val):
    for other in sol[sq].replace(val, ''):
        eliminate(sol, sq, other)

def eliminate(sol, sq, val):
    sol[sq] = sol[sq].replace(val, '')
    if len(sol[sq]) == 1:
        last = sol[sq][0]
        for p in peers[sq]:
            eliminate(sol, p, last)
    for u in units[sq]:
        candidates = [s for s in u if val in sol[s]]
        if len(candidates) == 1:
            assign(sol, candidates[0], val)
```

```
def assign(sol, sq, val):
    for other in sol[sq].replace(val, ''):
        eliminate(sol, sq, other)

def eliminate(sol, sq, val):
    # need to stop the recursion!
    if val not in sol[sq]:
        return
    sol[sq] = sol[sq].replace(val, '')
    if len(sol[sq]) == 1:
        last = sol[sq][0]
        for p in peers[sq]:
            eliminate(sol, p, last)
    for u in units[sq]:
        candidates = [s for s in u if val in sol[s]]
        if len(candidates) == 1:
            assign(sol, candidates[0], val)
```

Demo

```
def search(sol):
    # find a square with the least number of values > 1
    #
    # (1) "guess" one of the values (assign it) and
    #     try to solve the rest of the puzzle
    #
    # (2) if we guessed wrong and the solution is broken,
    #     backtrack --- i.e., restore the original values
    #     and guess another value
```

```
def search(sol):
    sq = min((s for s in squares if len(sol[s]) > 1),
             key=lambda s: len(sol[s]))
    # (1) "guess" one of the values (assign it) and
    #     try to solve the rest of the puzzle
    #
    # (2) if we guessed wrong and the solution is broken,
    #     backtrack --- i.e., restore the original values
    #     and guess another value
```

```
def search(sol):
    sq = min((s for s in squares if len(sol[s]) > 1),
             key=lambda s: len(sol[s]))
    assign(sol, sq, val)
    search(sol)

    # (2) if we guessed wrong and the solution is broken,
    #     backtrack --- i.e., restore the original values
    #     and guess another value
```

```
def search(sol):
    sq = min((s for s in squares if len(sol[s]) > 1),
             key=lambda s: len(sol[s]))

    orig = sol.copy() # copy the original grid
    for val in orig[sq]:
        assign(sol, sq, val)
        if not search(sol): # if search fails,
            sol.update(orig) #   restore original
        else:
            break
```

```
def search(sol):
    sq = min((s for s in squares if len(sol[s]) > 1),
             key=lambda s: len(sol[s]))

    orig = sol.copy() # copy the original grid
    for val in orig[sq]:
        assign(sol, sq, val)
        if not search(sol): # if search fails,
            sol.update(orig) #   restore original
        else:
            break
```

missing: search success/failure detection

```
def search(sol):
    if any(not sol[s] for s in squares):
        # a square with no values is illegal
        return False
    elif all(len(sol[s]) == 1 for s in squares):
        # if all squares have just one value, we're done
        return True
    else:
        sq = min((s for s in squares if len(sol[s]) > 1),
                 key=lambda s: len(sol[s]))
        orig = sol.copy()
        for val in orig[sq]:
            assign(sol, sq, val)
            if not search(sol):
                sol.update(orig)
            else:
                break
```

```
def search(sol):
    if any(not sol[s] for s in squares):
        return False
    elif all(len(sol[s]) == 1 for s in squares):
        return True
    else:
        sq = min((s for s in squares if len(sol[s]) > 1),
                 key=lambda s: len(sol[s]))
        orig = sol.copy()
        for val in orig[sq]:
            assign(sol, sq, val)
            if search(sol):
                return True # propagate success back up
            else:
                sol.update(orig)
        else:
            return False # no guesses worked = earlier fail
```

Demo

§Classes and OOP

```
class Point:
    pass

p = Point()
p.x = 10
p.y = 20
print('x=', p.x, 'y=', p.y)
```

```
x= 10 y= 20
```

```
class Point:
    def __init__(self): # `self` refers to "this" object
        self.x = 10
        self.y = 20

p = Point()
print('x=', p.x, 'y=', p.y)
```

```
x= 10 y= 20
```

```
class Point:
    def __init__(self, xinit, yinit):
        self.x = xinit
        self.y = yinit

p = Point(15, 25)
print('x=', p.x, 'y=', p.y)
```

```
x= 15 y= 25
```

```
class Point:
    ...
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def dist_to_origin(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def __str__(self): # automatically called by `print`
        return '({},{})'.format(self.x, self.y)

p = Point(0, 0)
p.translate(3, 4)
print('dist of', p, '=', p.dist_to_origin())
```

```
dist of (3,4) = 5.0
```

```
class Point:
    ...
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def dist_to_origin(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def __str__(self): # automatically called by `print`
        return '({},{})'.format(self.x, self.y)

p = Point(0, 0)
Point.translate(p, 4, 3) # same as p.translate(4, 3)
print('dist of', p, '=', Point.dist_to_origin(p))
```

```
dist of (4,3) = 5.0
```

```
class Point:
    ...
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def dist_to_origin(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def __str__(self): # automatically called by `print`
        return '({},{})'.format(self.x, self.y)

q = Point(0, 0)
r = Point(0, 0)
print(q == r)  # based on object identity (reference)
```

```
False
```

```
class Point:
    ...
    def __eq__(self, other): # called for '=='
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        return NotImplemented

p = Point(10, 20)
q = Point(10, 20)
r = Point(30, 40)

print(p == q)
print(p == r)
print(p == 'hello')
```

```
True
False
False
```

```
class Point:
    ...
    def __add__(self, other): # called for '+'
        if isinstance(other, Point):
            return Point(self.x+other.x, self.y+other.y)
        return NotImplemented

p = Point(10, 20)
q = Point(10, 20)
r = Point(30, 40)

print(p + q + r)
print(p + 10)
```

```
(50,80)
Traceback (most recent call last):
  File "classes.py", line 95, in <module>
    print(p+10)
TypeError: unsupported operand type(s) for +: 'Point' and 'int'
```

```
class Point:
    ...
    def __iter__(self):
        yield self.x # ok to not understand this now!
        yield self.y # ditto this

p = Point(19, 20)
for coord in p:
    print(coord)

l = list(p)
print(l)
cs = [c for c in p]
print(cs)
```

```
19
20
[19, 20]
[19, 20]
```

```
class Point:
    prev_coords = [] # class variable!

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        self.prev_coords.append((self.x, self.y))
        self.x += dx
        self.y += dy

p = Point()
q = Point(1, 2)

p.translate(5, 5)
p.translate(-3, 3)
print(p.prev_coords)
q.translate(10, 10)
print(q.prev_coords)
```

```
[(0, 0), (5, 5)]
[(0, 0), (5, 5), (1, 2)]
```

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.prev_coords = []  # instance variable

    def translate(self, dx, dy):
        self.prev_coords.append((self.x, self.y))
        self.x += dx
        self.y += dy


p = Point()
q = Point(1, 2)

p.translate(5, 5)
p.translate(-3, 3)
print(p.prev_coords)
q.translate(10, 10)
print(q.prev_coords)
```

```
[(0, 0), (5, 5)]
[(1, 2)]
```

```
class Point3D(Point): # inherit from Point
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def dist_to_origin(self):
        dist2d = super().dist_to_origin()
        return sqrt(dist2d ** 2 + self.z ** 2)

    def __str__(self): # override base class impl
        return '({},{},{})'.format(self.x, self.y, self.z)

p = Point3D(1, 1, 1)
print('dist of', p, '=', p.dist_to_origin())

p.translate(10, 10)
p.translate(10, 10)
print(p)
print(p.prev_coords)
print(p + Point(5, 5))
```

```
dist of (1,1,1) = 1.732
(21,21,1)
[(1, 1), (11, 11)]
(26,26)
```