

# Tuples and lists

- tuples
- lists
- mutability
- list comprehension
- for-if, for-for
- list()
- any(), all()
- enumerate(), zip()

# Tuples

$(\text{value}_0, \text{ value}_1, \dots, \text{ value}_{k-1})$

- Tuples can contain a sequence of zero or more elements, enclosed by "()" and ")"
- Tuples are **immutable**
- Tuple of length 0: ()
- Tuple of length 1: (value,)  
Note the **comma** to make a tuple of length one distinctive from an expression in parenthesis
- In many contexts a tuple with  $\geq 1$  elements can be written without parenthesis
- Accessors to lists also apply to tuples, slices, ...

## Python shell

```
> (1, 2, 3)
| (1, 2, 3)
> ()
| ()
> (42) !!!!!
| 42
> (42,)
| (42,)
> 1, 2
| (1, 2)
> 42,
| (42,)
> x = (3, 7)
> x
| (3, 7)
> x = 4, 6
> x
| (4, 6)
> x[1] = 42
| TypeError: 'tuple' object does
| not support item assignment
```

# Question – What value is $((42,))$ ?

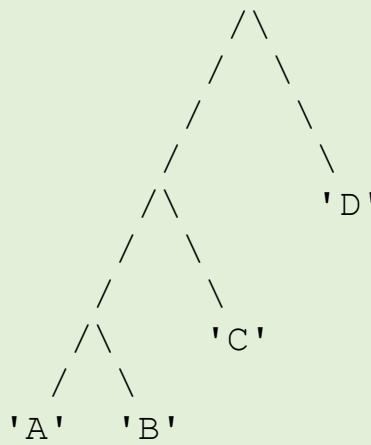
- a) 42
- b)  $(42)$
-  c)  $(42,)$
- d)  $((42,),)$
- e) Don't know

# Question – What is $x$ ?

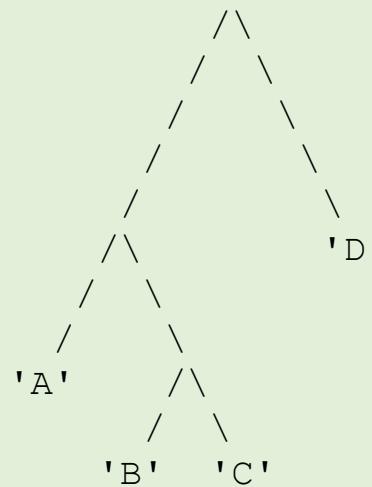
```
x = [1, [2, 3], (4, 5)]  
x[2][0] = 42
```

- a) [1, [42, 3], (4, 5)]
- b) [1, [2, 3], (42, 5)]
- c) [1, [2, 3], 42]
-  d) TypeError
- e) Don't know

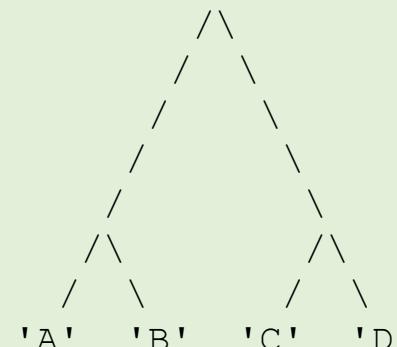
# Question – What tree is ('A', (( 'B' , 'C' ) , 'D' ) ) ?



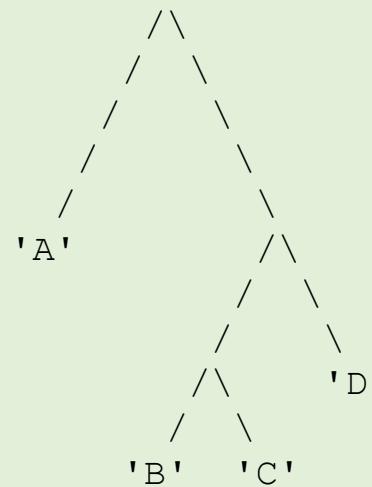
a)



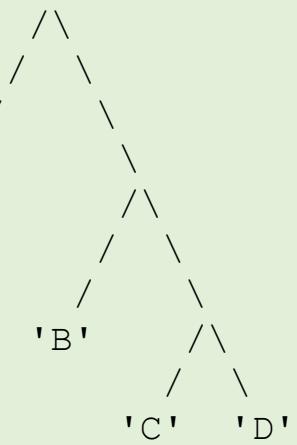
b)



c)



d)



e)

f) Don't know

# Tuple assignment

## Python shell

```
> point = (10, 25)
> x, y = point
> x
| 10
> y
| 25
```

- Parallel assignments

$$x, y, z = a, b, c$$

is a short hand for a tuple assignment (right side is a single tuple)

$$(x, y, z) = (a, b, c)$$

- First the right-hand side is evaluated completely, and then the individual values of the tuple are assigned to  $x, y, z$  left-to-right (length must be equal on both sides)

# Nested tuple/lists assignments

- Let hand side can be nested  
(great for unpacking data)

```
(x, (y, (a[0], w)), a[1])  
= 1, (2, (3, 4)), 5
```

- [...] and (...) on left side matches both lists and tuples of equal length  
(but likely you would like to be consistent with type of parenthesis)

## Python shell

```
> two_points = [(10, 25), (30, 40)]  
> (x1, y1, x2, y2) = two_points  
| ValueError: not enough values to  
unpack (expected 4, got 2)  
> ((x1, y1), (x2, y2)) = two_points  
> a = [None, None]  
> v = ((2, (3, 4)), 5)  
> ((y, (a[0], w)), a[1]) = v  
> a  
| [3, 5]  
> [x, y, z] = (3, 5, 7)  
> (x, y, z) = [3, 5, 7]  
> [x, (y, z), w] = (1, [2, 3], 4)  
> [x, (y, z), w] = (1, [2, (5, 6)], 4)  
> z  
| (5, 6)
```

# Tuples vs lists: $a += b$

- Lists

Extends existing list, i.e. same as `a.extend(b)`

- Tuples

Must create a new tuple `a + b` and assign to `a`  
(since tuples are immutable)

## Python shell

```
> (1, 2) + (3, 4)
| (1, 2, 3, 4)
> x = [1, 2]
> y = x
> y += [3, 4]
> y
| [1, 2, 3, 4]
> x
| [1, 2, 3, 4] !
```

```
> x = (1, 2)
> y = x
> y += (3, 4)
> y
| (1, 2, 3, 4)
> x
| (1, 2) !
```

# \*variable assignment

- For a tuple of variable length a single *\*variable name* on the left side will be assigned a list of the remaining elements not matched by variables preceding/following \*
- Example

```
a, *b, c = t
```

is equivalent to

```
a = t[0]
```

```
b = t[1:-1]
```

```
c = t[-1]
```

- There can be a single \* in a left-hand-side tuple (but one new \* in each nested tuple)

## Python shell

```
> (a, *b, c, d) = (1, 2, 3, 4, 5, 6)
> b
| [2, 3, 4]
> (a, *b, c, d) = (1, 2, 3)
> b
| []
> (a, *b, c, d) = (1, 2)
| ValueError: not enough values to
| unpack (expected at least 3, got 2)
> v = ((1,2,3),4,5,6,(7,8,9,10))
> ((a, *b), *c, (d, *e)) = v
> b
| [2, 3]
> c
| [4, 5, 6]
> e
| [8, 9, 10]
> head, *tail = [1, 2, 3, 4]
> head
| 1
> tail
| [2, 3, 4]
```

# Question – What is b ?

`(*a, (b, ), c) = ((1, 2), ((3, 4)), ((5, )), (6))`

a) (1, 2)

b) (3, 4)

 c) 5

d) (5, )

e) (6)

f) Don't know

## Python shell

```
> (*a, (b, ), c) = ((1,2), ((3,4)), ((5,)), (6))
> a
| [(1, 2), (3, 4)]
> b
| 5
> c
| 6
```

# \* in list and tuple construction

- When constructing a list or tuple you can insert zero or more elements from another list/tuple/sequence by inserting *\*expression*
- There can be an arbitrary number of \* expressions in a tuple or list construction

## Python shell

```
> A = (1, 2, 3)
> B = ['B', 'C']
> L = [A, B, 4, 5]
> L
| [(1, 2, 3), ['B', 'C'], 4, 5]
> len(L)
| 4
> L = [*A, *B, 4, 5]
> L
| [1, 2, 3, 'B', 'C', 4, 5]
> len(L)
| 7
> (*A, *B, 4, 5)
| (1, 2, 3, 'B', 'C', 4, 5)
```

## Python shell

```
> from timeit import timeit
> timeit('A + A + A + A + A + A + A + A', setup='A = [1,2,3,4,5,6,7,8,9,10]')
| 0.665172699955292 # repeated concatenation can be slow
!> timeit('[*A, *A, *A, *A, *A, *A, *A, *A]', setup='A = [1,2,3,4,5,6,7,8,9,10]')
| 0.32599859999027103 # * notation can be faster for multiple concatenation
```

# List comprehension (cool stuff)

- Example:

```
[x*x for x in [1, 2, 3]]
```

returns

```
[1, 4, 9]
```

- General

*[expression for variable in sequence]*

returns a list, where *expression* is computed for each element in *sequence* assigned to *variable*

## Python shell

```
> [2*x for x in [1,2,3]]
| [2, 4, 6]
> [2*x for x in (1,2,3)]
| [2, 4, 6]
> [2*x for x in range(10,15)]
| [20, 22, 24, 26, 28]
> [2*x for x in 'abc']
| ['aa', 'bb', 'cc']
> [(None, None) for _ in range(2)]
| [(None, None), (None, None)]
```

# List comprehension (it's just syntactic sugar...)

Python shell

```
> [x*2 for x in [1, 2, 3]]  
| [2, 4, 6]  
  
> L = []  
> for x in [1, 2, 3]:  
>     L.append(x*2)  
> L  
| [2, 4, 6]
```

# List comprehension (more cool stuff)

- Similarly to the left-hand-side in assignments, the variable part can be a (nested) tuple of variables for unpacking elements:

[*expression* **for** *tuple of variables* **in** *sequence*]

## Python shell

```
> points = [(3, 4), (2, 5), (4, 7)]
> [(x, y, x*y) for (x, y) in points]
| [(3, 4, 12), (2, 5, 10), (4, 7, 28)]
> [(x, y, x*y) for x, y in points]
| [(3, 4, 12), (2, 5, 10), (4, 7, 28)]
> [x, y, x*y for (x, y) in points]
| SyntaxError: invalid syntax
```



parenthesis required for  
the constructed tuples

# List comprehension – for-if and multiple for

- List comprehensions can have nested for-loops

[*expression* for  $v_1$  in  $s_1$  for  $v_2$  in  $s_2$  for  $v_3$  in  $s_3$ ]

- Can select a subset of the elements by adding an if-condition

[*expression* for  $v_1$  in  $s_1$  if *condition*]

- and be combined...

## Python shell

```
> [(x, y) for x in range(1, 3) for y in range(4, 6)]
| [(1, 4), (1, 5), (2, 4), (2, 5)]
> [x for x in (1, 2) for x in (4, 5)] !!!!!
| [4, 5, 4, 5]
> [x for x in range(1, 101) if x % 7 == 1 and x % 5 == 2]
| [22, 57, 92]
> [(x, y, x*y) for x in range(1, 11) if 6 <= x <= 7 for y in range(x, 11) if 6 <= y <= 7 and not x == y]
| [(6, 7, 42)]
```

# Question – What will print the same?

```
points = [ (3, 7), (4, 10), (12, 3), (9, 11), (7, 5) ]  
print([ (x, y) for x, y in points if x < y ])
```

- a) `print([x, y for x, y in points if x < y] )`
- b) `print([(x, y) for p in points if p[0] < p[1]] )`
- c) `print([p for p in points if p[0] < p[1]] )`
- d) `print([[x, y] for x, y in points if x < y] )`
- e) Don't know

# any, all

- `any(L)` checks if at least one element in the sequence L is true (list, tuple, ranges, sequence, strings, ...)

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

- `all(L)` checks if all elements in the sequence L are true

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

- `any` and `all` return `True` or `False`

## Python shell

```
> any((False, True, False))
| True
> any([False, False, False])
| False
> any([])
| False
> all([False, False, True])
| False
> all((True, True, True))
| True
> all(())
| True
> L = (7, 42, 13)
> any([x == 42 for x in L])
| True
> all([x == 42 for x in L])
| False
```

# Example – computing primes

Python shell

```
> [x for x in range(2, 50) if all([x % f for f in range(2, x)])]
| [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

> [10 % f for f in range(2, 10)]
| [0, 1, 2, 0, 4, 3, 2, 1]
> all([10 % f for f in range(2, 10)]) # == 0 is considered False
| False
> [13 % f for f in range(2, 13)]
| [1, 1, 1, 3, 1, 6, 5, 4, 3, 2, 1]
> all([13 % f for f in range(2, 13)])
| True
```

# enumerate

```
list(enumerate(L))
```

returns

```
[ (0, L[0]), (1, L[1]), ..., (len(L) - 1, L[-1]) ]
```

## Python shell

```
> points = [(1, 2), (3, 4), (5, 6)]
> [(idx, x * y) for idx, (x, y) in enumerate(points)]
| [(0, 2), (1, 12), (2,30)]

> L = ('a', 'b', 'c')
> list(enumerate(L))
| [(0, 'a'), (1, 'b'), (2, 'c')]

> L_ = []
> for idx in range(len(L)):
>     L_.append((idx, L[idx]))
> print(L_)
| [(0, 'a'), (1, 'b'), (2, 'c')]
> list(enumerate(['a', 'b', 'c'], start=7))
| [(7, 'a'), (8, 'b'), (9, 'c')]
```

# zip

`list(zip(L1, L2, ..., Lk)) = [(L1[0], L2[0], ..., Lk[0]), ..., (L1[n-1], L2[n-1], ..., Lk[n-1])]`  
where  $n = \min(\text{len}(L_1), \text{len}(L_2), \dots, \text{len}(L_k))$

- Example (“matrix transpose”):

```
list(zip([1, 2, 3],  
         [4, 5, 6],  
         [7, 8, 9]))
```

returns

```
[ (1, 4, 7),  
  (2, 5, 8),  
  (3, 6, 9) ]
```

## Python shell

```
> x = [1, 2, 3]  
> y = [4, 5, 6]  
| zip(x, y)  
> <zip at 0xb02b530>  
> points = list(zip(x, y))  
> print(points)  
| [(1, 4), (2, 5), (3, 6)]
```

## Python shell

```
> first = ['Donald', 'Mickey', 'Scrooge']
> last = ['Duck', 'Mouse', 'McDuck']

> for i, (a, b) in enumerate(zip(first, last), start=1):
>     print(i, a, b)

| 1 Donald Duck
| 2 Mickey Mouse
| 3 Scrooge McDuck
```

# (Simple) functions

- You can define your own functions using:

```
def function-name (var1, ..., vark):  
    body code
```

- If the body code executes  
 return *expression*

the result of *expression* will be returned by the function. If expression is omitted or the body code terminates without performing return, then None is returned.

- When calling a function *name* (*value*<sub>1</sub>, ..., *value*<sub>*k*</sub>) body code is executed with *var*<sub>*i*</sub>=*value*<sub>*i*</sub>

## Python shell

```
> def sum3(x, y, z):  
    return x + y + z  
  
> sum3(1, 2, 3)  
| 6  
> sum3(5, 7, 9)  
| 21  
  
> def powers(L, power):  
    P = [x**power for x in L]  
    return P  
  
> powers([2, 3, 4], 3)  
| [8, 27, 64]
```

# Question – What tuple is printed ?

```
def even(x):  
    if x % 2 == 0:  
        return True  
    else:  
        return False
```

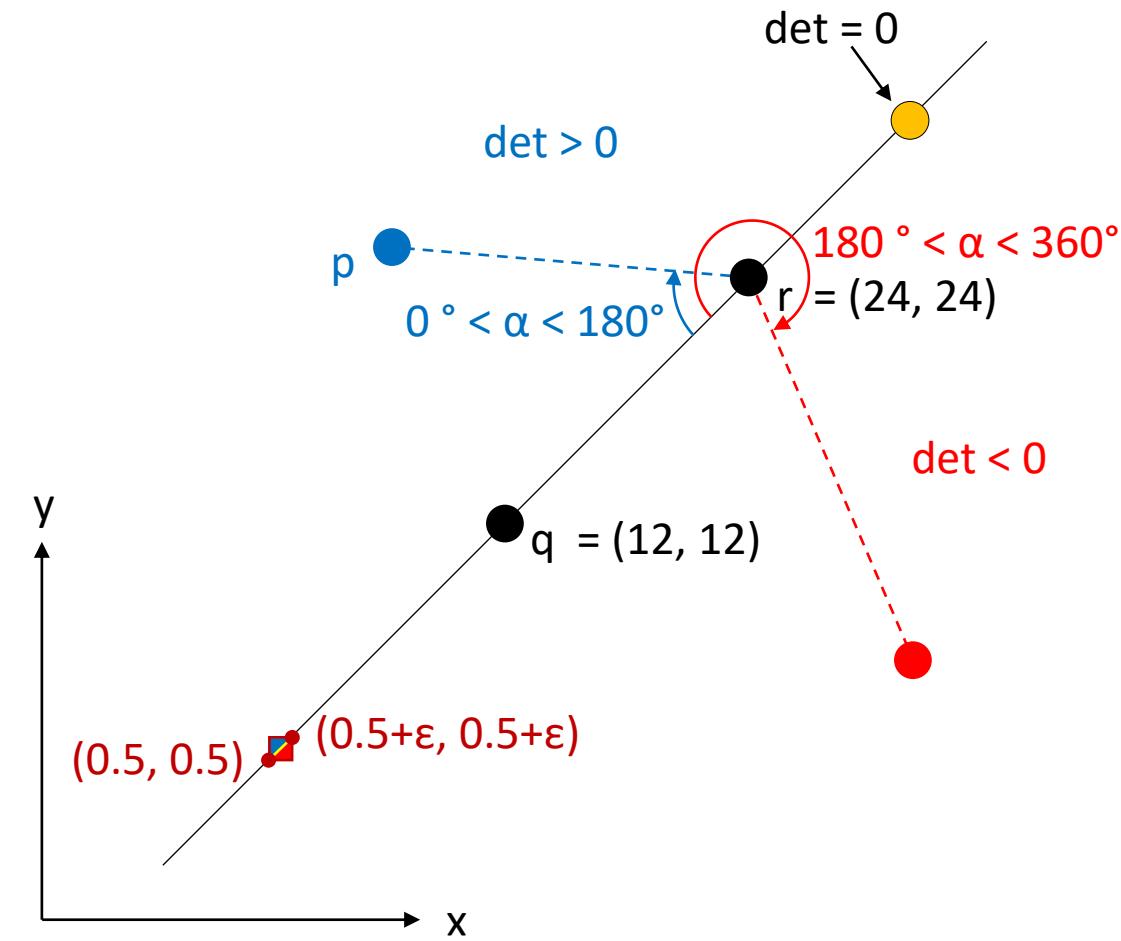
```
print( (even(7), even(6)) )
```

- a) (False, False)
-  b) (False, True)
- c) (True, False)
- d) (True, True)
- e) Don't know

# Geometric orientation test

## Purpose of example

- illustrate tuples
- list comprehension
- matplotlib.pyplot
- floats are strange



$$\det = \begin{vmatrix} 1 & q_x & q_y \\ 1 & r_x & r_y \\ 1 & p_x & p_y \end{vmatrix} = \underbrace{r_x p_y - p_x r_y - q_x p_y + p_x q_y + q_x r_y - r_x q_y}_{6! = 720 \text{ different orders to add}}$$

$6! = 720$  different orders to add



## sign-plot.py

```
import matplotlib.pyplot as plt

N = 256
delta = 1 / 2**54
q = (12, 12)
r = (24, 24)
P = [] # points (i, j, det)

for i in range(N):
    for j in range(N):
        p = (1/2 + i * delta, 1/2 + j * delta)
        det = (q[0]*r[1] + r[0]*p[1] + p[0]*q[1]
               - r[0]*q[1] - p[0]*r[1] - q[0]*p[1])
        P.append((i, j, det))

pos = [(i, j) for i, j, det in P if det > 0]
neg = [(i, j) for i, j, det in P if det < 0]
zero = [(i, j) for i, j, det in P if det == 0]

plt.subplot(facecolor='lightgrey', aspect='equal')
plt.xlabel('i')
plt.ylabel('j', rotation=0)

for points, color in [(pos, "b") , (neg, "r") , (zero, "y")]:
    X = [x for x, y in points]
    Y = [y for x, y in points]
    plt.plot(X, Y, color + ".")  
plt.plot([-1, N], [-1, N], "k-")
plt.show()
```

