Python-part1

May 17, 2024

1 Introduction to Python

```
[1]: print("hello world", "!")
```

```
hello world !
```

```
[2]: # this is a comment
```

1.1 Basic Types and Operations

1.1.1 Integers

```
[3]: 42
```

```
[3]: 42
```

```
[4]: 2*3*7
```

```
[4]: 42
```

```
[5]: type(42)
```

```
[5]: int
```

```
[6]: 2 ** 4 # exponentiation
```

```
[6]: 16
```

```
[7]: 2 ** 3456 # int is arbitrarily large
```

```
[7]: 228910131428177603701755406038597370992866256605996714416239305740707356279544553
     4171714834328144730325654782035422244494225444277671033999347942675541124921525
     7064374778361539328935788158075188729160957705048611938593611612558140447042909
     7773717389138692933902315347725871293766854763709974986644983264429564711070350
     5635220403991381845027655795652717764375247672196073643303436202463972523755903920
     6709931369552473891279007543186250320841012395519779084490236891515654306686922
     8757701257705464810732957386428986999720416931970101723139791242081350688640076
     6112131847874647236048907022700217138958296139692769949850143644876782633083624900
     385903354236106269801910000054731259215974179239403248038668469048122459970394633
```

```
02644995379033586738360423904808127237636449988915417461938128613213972680254190
44221533969071895396348012188498227483764751408143904811379716106366515435388997
32162561295745248744175743178153962997058705210024539453107059157058925528424545
39325750759736499843028430365261853724137109257425954039503707968037797834103193
6
```

```
[8]: type(2 ** 3456)
```

```
[8]: int
```

```
[9]: 6//7 # integer division returns an integer (discards the fractional part)
```

```
[9]: 0
```

```
[10]: 6%7
```

```
[10]: 6
```

1.1.2 Booleans

```
[11]: True
```

```
[11]: True
```

```
[12]: False
```

```
[12]: False
```

```
[13]: True and False
```

```
[13]: False
```

```
[14]: True or False
```

```
[14]: True
```

```
[15]: 42 == 41
```

```
[15]: False
```

```
[16]: 17 <= 41
```

```
[16]: True
```

```
[17]: True + 1
```

```
[17]: 2
```

```
[18]: False + 1
```

```
[18]: 1
```

```
[19]: True == 1
```

```
[19]: True
```

Hint!

True and False can also be treated as the integers 1 and 0, respectively. They are converted to numbers automatically.

1.1.3 Floating Point Numbers

```
[20]: type(3.14)
```

```
[20]: float
```

```
[21]: 6/7 # division returns a floating point number
```

```
[21]: 0.8571428571428571
```

```
[22]: 1/10
```

```
[22]: 0.1
```

```
[23]: 0.1 == 0.1000000000000000005551115
```

```
[23]: True
```

Why???

Floating point numbers are represented in hardware as binary (base-2) fractions.

The actual stored value is the nearest representable binary fraction. In the case of 1/10, the hardware representation is close to but not exactly equal to the true value of 1/10

1.2 Explicit Type Conversions (Casting)

```
[24]: int(2.99999)
```

```
[24]: 2
```

```
[25]: float(42)
```

```
[25]: 42.0
```

```
[26]: str(42)
```

```
[26]: '42'
```

1.3 Variables and Assignment

Variables in Python are not explicitly declared!

[27]:
```
a = 42
```

[28]:
```
print(a)
```

```
42
```

[29]:
```
a = b = 42
```

[30]:
```
print(a,b)
```

```
42 42
```

[31]:
```
a, b = 17, 42
```

[32]:
```
print(a,b)
```

```
17 42
```

[33]:
```
a, b = 17, 42
a, b = b, a
print(a, b)
```

```
42 17
```

1.4 For Loops

[34]:
```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[35]:
```
for i in range(10):
    print(i)
    print(i+1)
```

```
0
1
1
```

```
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
```

[36]:
```python
for i in range(10):
    print(i)
print(i+1)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

White spaces **DO** matter! - Python uses indentation to indicate a block of code - Use 4 spaces per indentation level (style guide) - Avoid the use of tabs

[37]:
```python
for i in range(3,10):
    print(i)
```

```
3
4
5
6
7
8
9
```

[38]:
```
for i in range(3,10,4):
    print(i)
```

```
3
7
```

[39]:
```
for i in range(10,0,-1):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
```

[40]:
```
for i in [42, "dogs", "and", "cats"]:
    print(i)
```

```
42
dogs
and
cats
```

[41]:
```
for x in "hello world": print(x)
```

```
h
e
l
l
o

w
o
r
l
d
```

1.5 Function Definitions

[42]:
```
def inc(x):
    return x + 1
```

[43]:
```
print(inc(41))
```

```
42
```

[44]:
```python
def fib(n):
    """ This is the fibonacci function! """
    if n == 0: return 0
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a + b
    return b
```

[45]:
```python
help(fib)
```

```
Help on function fib in module __main__:

fib(n)
    This is the fibonacci function!
```

[46]:
```python
print(fib(5))
```

```
5
```

[47]:
```python
print(fib(6))
```

```
8
```

[48]:
```python
print(fib(7))
```

```
13
```

1.6 Function Arguments

[49]:
```python
a, b, c = 17, 42, fib(11)
print(a,b,c)
```

```
17 42 89
```

[50]:
```python
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
```

```
        whether to forcibly flush the stream.
```

Python supports functions with: - Variable number of arguments (like `*args` above), wrapped in a tuple. - Optional arguments, with default values (like `sep=' '` and `end='\n'`)

[51]:
```
print(a, b, c, sep = " and ")
```

```
17 and 42 and 89
```

[52]:
```
print(c, b, a, end="") # do not add a new line at the end
print("!")
```

```
89 42 17!
```

[53]:
```
print(c, b, a, end=".", sep=", ")
```

```
89, 42, 17.
```

We can define a fibonacci function with custom start values:

[54]:
```
def fib(n, start_a=0, start_b=1):
    if n == 0: return start_a
    a, b = start_a, start_b
    for i in range(n-1):
        c = a + b
        a, b = b, c
    return b
```

[55]:
```
print(fib(7))
```

```
13
```

[56]:
```
print(fib(7, start_a=17, start_b=42))
```

```
682
```

- *Keyword arguments* in Python are passed using to function using their names and can be in arbitrary order.
- Must always come **after** positional arguments.

[57]:
```
print(fib(start_b=42, n=7, start_a=17))
```

```
682
```

[58]:
```
fib(8,"hello ","world! ")
```

[58]:
```
'hello world! world! hello world! world! hello world! hello world! world! hello
world! world! hello world! hello world! world! hello world! hello world! world!
hello world! world! hello world! hello world! world! hello world! '
```

Hint!

`+` is overloaded and if its arguments are stings it denotes concatenation.

1.7 Namespaces and Scopes

- A *namespace* maps names to values.
- Different namespaces have different lifetimes.
- Examples of namespaces:
  - The global names in a module.
  - The local names in a function invocation.
  - the set of built-in names (e.g., `range()`, `abs()`).
- *Scope* is the textual region where a name is visible.

[59]:
```
a = 42

def reasonable():
    print(a)

reasonable()
print(a)
```

```
42
42
```

[60]:
```
a = 42

def what():
    a = 17  # this is a local variable, which is different from the global `a`
    print(a)

what()
print(a)
```

```
17
42
```

When `what` is active, there are three active scopes: - The local scope (containing the local name `a`) - The global scope (containing the global name `a`) - The outermost scope containing built-in names

The previous program is equivalent to this:

[61]:
```
a = 42

def what():
    b = 17
    print(b)

what()
print(a)
```

```
17
42
```

[62]:
```
a = 42

def what():
    global a
    a = 17 # now a refers to the global a
    print(a)

what()
print(a)
```

```
17
17
```

When a name is declared `global`, then all references to it go directly to the global scope.

[63]:
```
a = 42

def what():
    global a # this only refers to the variable in the current local scope
    a = 17 # a is global
    def what2():
        a = 11 # a is local
        print(a)
    what2()
    print(a)

what()
print(a)
```

```
11
17
17
```

1.8 Sequences

1.8.1 Strings

[64]:
```
"This is a string"
```

[64]:
```
'This is a string'
```

[65]:
```
type("This is a string")
```

[65]:
```
str
```

[66]:
```
'This is also a string'
```

[66]:
```
'This is also a string'
```

```
[67]: """This is a
      multiline
      string"""
```

```
[67]: 'This is a\nmultiline\nstring'
```

```
[68]: print("""This is a
      multiline
      string""")
```

```
This is a
multiline
string
```

Lexicographic comparison:

```
[69]: "abc" < "acb"
```

```
[69]: True
```

```
[70]: "42" + " is the answer to life the universe and everything" # string␣
       ↪concatenation
```

```
[70]: '42 is the answer to life the universe and everything'
```

Strings are **immutable**!

```
[71]: s = "This is a string"
      s[3] = 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[71], line 2
      1 s = "This is a string"
----> 2 s[3] = 'a'

TypeError: 'str' object does not support item assignment
```

Indexing and Slicing

```
[ ]: s = "This is a string"
     s[3]
```

Access time is O(1)

```
[ ]: s[3:]
```

```
[ ]: print(s)
```

[ ]:
```
s[:7]
```

[ ]:
```
s[3:7]
```

Hint!

Returns the index range [3,7)

[ ]:
```
s[3:7:2]
```

[ ]:
```
s = s[:6] + 'z' + s[7:]
s
```

Hint!

Doesn't change the initial string, but creates a new string object.

[ ]:
```
s = "This is a string"
s[6:2:-1] # From index 6 (included) to index 2 (not included) with step -1
```

[ ]:
```
s[::-1] # From the end to the beginning with step -1 (i.e. reverse)
```

[ ]:
```
s = "0123456789"
print(s[::3])
```

Hint!

Pick every item that's a multiple of 3.

[ ]:
```
s = "..0123456789.."
print(s[2::3])
```

Hint!

Pick every item that's a multiple of 3, starting from index 2.

Slice operations create a **new** string object!

Slice complexity?

O(n)

[ ]:
```
"this is a string".upper()
```

1.8.2 Tuples

[ ]:
```
(1,2,3)
```

[ ]:
```
t = (42, "cats", True, "dogs")
t
```

[72]:
```
t[0]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[72], line 1
----> 1 t[0]

NameError: name 't' is not defined
```

Hint!

Tuples are zero-indexed.

```
[ ]: t[1]
```

Indexes of tuples can be **dynamic**!!!

```
[73]: for i in [0,1,2]:
          print(t[i])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[73], line 2
      1 for i in [0,1,2]:
----> 2     print(t[i])

NameError: name 't' is not defined
```

Why is this not allowed in ML?

In ML the tuple index must be static so that the type of the projected element is statically known.

```
[ ]: t[0] = 43
```

Tuples are also **immutable**!

```
[74]: t
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[74], line 1
----> 1 t

NameError: name 't' is not defined
```

Slices work just as in strings:

```
[75]: t[1:3]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[75], line 1
----> 1 t[1:3]

NameError: name 't' is not defined
```

[76]:
```
t[::-1]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[76], line 1
----> 1 t[::-1]

NameError: name 't' is not defined
```

[77]:
```
t = (1, (42, 17), True, "string")
t[1][0]
```

[77]:
```
42
```

How can we access 42?

[78]:
```
(1,2)+(3,4)
```

[78]:
```
(1, 2, 3, 4)
```

[79]:
```
1,2 + 3,4
```

[79]:
```
(1, 5, 4)
```

[80]:
```
t + ("is this a singleton tuple?")
# the same as t + "is this a singleton tuple?"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[80], line 1
----> 1 t + ("is this a singleton tuple?")
      2 # the same as t + "is this a singleton tuple?"

TypeError: can only concatenate tuple (not "str") to tuple
```

[81]:
```
t + ("this is a singleton tuple",)
```

[81]:
```
(1, (42, 17), True, 'string', 'this is a singleton tuple')
```

```
[82]: t + () # the empty tuple
```

```
[82]: (1, (42, 17), True, 'string')
```

```
[83]: ("my tuple", 42) * 3
```

```
[83]: ('my tuple', 42, 'my tuple', 42, 'my tuple', 42)
```

Lexicographic Comparison

```
[84]: (1, 2, 3, 4) < (1, 2, 5)
```

```
[84]: True
```

1.8.3 Lists

```
[85]: l = [1, 2, 3, 4]
      l
```

```
[85]: [1, 2, 3, 4]
```

```
[86]: l = [1, 2, "string", 3.14, 4]
      l
```

```
[86]: [1, 2, 'string', 3.14, 4]
```

```
[87]: l = list(range(1,5))
      print(l)
```

```
[1, 2, 3, 4]
```

Slices and indexing work just as in strings and tuples:

```
[88]: l = [1, 2, "string", 3.14, 4]
      l[3]
```

```
[88]: 3.14
```

```
[89]: i = 3
      l[i]
```

```
[89]: 3.14
```

```
[90]: l[42]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[90], line 1
----> 1 l[42]
```

```
IndexError: list index out of range
```

[91]:
```
l = [1, 2, "string", 3.14, 4]
l[2:4]
```

[91]:
```
['string', 3.14]
```

[92]:
```
l[::-1]
```

[92]:
```
[4, 3.14, 'string', 2, 1]
```

Lists **are muttable**!!

[93]:
```
l[1] = [3,4,5]
print(l)
```

```
[1, [3, 4, 5], 'string', 3.14, 4]
```

[94]:
```
del l[1]
l
```

[94]:
```
[1, 'string', 3.14, 4]
```

- Accessing is O(1)
- `del` is O(n)

[95]:
```
l.append(4)
l
```

[95]:
```
[1, 'string', 3.14, 4, 4]
```

[96]:
```
l.pop()
```

[96]:
```
4
```

Lists can be used as stacks!

[97]:
```
l2d = [[1,2,3],[4,5,6]]
l2d
```

[97]:
```
[[1, 2, 3], [4, 5, 6]]
```

[98]:
```
l2d[0][1]
```

[98]:
```
2
```

[99]:
```
l1 = [1,2,3]
l = l1
```

```
[100]: l2 = [l1, l, l]
```

```
[101]: l2[0][1] = 42
```

```
[102]: l2
```

```
[102]: [[1, 42, 3], [1, 42, 3], [1, 42, 3]]
```

```
[103]: l2 = [l1,l1,l1]
       l2
```

```
[103]: [[1, 42, 3], [1, 42, 3], [1, 42, 3]]
```

```
[104]: l2[0][1]=42
```

```
[105]: l2
```

```
[105]: [[1, 42, 3], [1, 42, 3], [1, 42, 3]]
```

```
[106]: l1
```

```
[106]: [1, 42, 3]
```

```
[107]: l3 = [l1[:], l1[:], l1[:]] # l1[:] is a slice operation and creates a new object
       l3
```

```
[107]: [[1, 42, 3], [1, 42, 3], [1, 42, 3]]
```

```
[108]: l3[0][1] = 43
       l3
```

```
[108]: [[1, 43, 3], [1, 42, 3], [1, 42, 3]]
```

```
[109]: l4 = [1,2,3]
       for x in l4: x = 0
```

```
[110]: l4
```

```
[110]: [1, 2, 3]
```

1.8.4 Conversions Between Sequence Types

```
[111]: list("abc")
```

```
[111]: ['a', 'b', 'c']
```

```
[112]: tuple(list("abc"))
```

```
[112]: ('a', 'b', 'c')
```

```
[113]: str([1,2,3])
```

```
[113]: '[1, 2, 3]'
```

1.9 Sets

```
[114]: s = {1, 2, 4, 6}
       s
```

```
[114]: {1, 2, 4, 6}
```

```
[115]: 2 in s
```

```
[115]: True
```

```
[116]: 3 in s
```

```
[116]: False
```

```
[117]: e = set() # the empty set
       e
```

```
[117]: set()
```

```
[118]: t = {0, 1, 4, 9, 16}
```

```
[119]: t.add(25)
       t
```

```
[119]: {0, 1, 4, 9, 16, 25}
```

```
[120]: t.remove(25)
       t
```

```
[120]: {0, 1, 4, 9, 16}
```

- In Python sets are implemented with *hash tables*
- lookup/insert/delete operations are O(1) on average.

```
[121]: len(t)
```

```
[121]: 5
```

```
[122]: print(t)
```

```
{0, 1, 16, 4, 9}
```

```
[123]: print(s)
```

```
{1, 2, 4, 6}
```

```
[124]: t & s # set intersection
```

```
[124]: {1, 4}
```

Complexity is O(min(len(t), len(s)))

```
[125]: t | s # set union
```

```
[125]: {0, 1, 2, 4, 6, 9, 16}
```

Complexity is O(len(t) + len(s))

```
[126]: t - s # set difference
```

```
[126]: {0, 9, 16}
```

Complexity is O(len(t))

`in` can be used for sequences as well:

```
[127]: 3 in [1,2,4,5]
```

```
[127]: False
```

```
[128]: 'h' in "hello world!"
```

```
[128]: True
```

But…

complexity is O(n)

```
[129]: t <= s # is subset?
```

```
[129]: False
```

```
[130]: {1,2,3} <= {1,2,3,4}
```

```
[130]: True
```

```
[131]: for x in {0,4,100,5}: print(x)
```

```
0
5
100
4
```

```
[132]: sorted({0,4,100,5})
```

```
[132]: [0, 4, 5, 100]
```

```
[133]: sorted((1,5,2))
```

```
[133]: [1, 2, 5]
```

Can also be used for sequences:

```
[134]: sorted([0,4,100,5])
```

```
[134]: [0, 4, 5, 100]
```

```
[135]: for x in sorted({0,4,100,5}): print(x)
```

```
0
4
5
100
```

```
[136]: s = {"cats", "dogs", 42, True}
```

```
[137]: sorted(s)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[137], line 1
----> 1 sorted(s)

TypeError: '<' not supported between instances of 'int' and 'str'
```

```
[138]: s = {"cats", "dogs", 42, True, (1,2)}
       s
```

```
[138]: {(1, 2), 42, True, 'cats', 'dogs'}
```

```
[139]: s = {"cats", "dogs", 42, True, {1,2}}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[139], line 1
----> 1 s = {"cats", "dogs", 42, True, {1,2}}

TypeError: unhashable type: 'set'
```

```
[140]: s = {"cats", "dogs", 42, True, [1,2]}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[140], line 1
----> 1 s = {"cats", "dogs", 42, True, [1,2]}
```

```
TypeError: unhashable type: 'list'
```

Why..?

Mutable objects cannot be added to a set!

Why..?

Because their hash value may change!

```
[ ]: f = frozenset({1,2,3,4}) # an immutable set
```

```
[141]: f.add(5)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[141], line 1
----> 1 f.add(5)

NameError: name 'f' is not defined
```

`frozenset` is immutable and can be added to a set

```
[142]: s = {"cats", "dogs", 42, True, frozenset({1,2})}
       s
```

```
[142]: {42, True, 'cats', 'dogs', frozenset({1, 2})}
```

1.10 Dictionaries

```
[143]: d = {} # the empty dictionary
```

```
[144]: d[1] = 42
       d[2] = 17
       d
```

```
[144]: {1: 42, 2: 17}
```

```
[145]: d = {1: 42, 2: 17}
       d
```

```
[145]: {1: 42, 2: 17}
```

```
[146]: d[3] = "cats"
       d["dog"] = "no cats"
```

```
[147]: d
```

```
[147]: {1: 42, 2: 17, 3: 'cats', 'dog': 'no cats'}
```

```
[148]: d[[1,2]] = "list [1, 2]" # indices must be hashable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[148], line 1
----> 1 d[[1,2]] = "list [1, 2]" # indices must be hashable

TypeError: unhashable type: 'list'
```

```
[149]: d["list"] = [1,2] # values need not be hashable
```

```
[150]: for key in d: print(key, "maps to", d[key])
```

```
1 maps to 42
2 maps to 17
3 maps to cats
dog maps to no cats
list maps to [1, 2]
```

```
[151]: list(d.keys())
```

```
[151]: [1, 2, 3, 'dog', 'list']
```

```
[152]: list(d.values())
```

```
[152]: [42, 17, 'cats', 'no cats', [1, 2]]
```

```
[153]: list(d.items())
```

```
[153]: [(1, 42), (2, 17), (3, 'cats'), ('dog', 'no cats'), ('list', [1, 2])]
```

```
[154]: for key,value in d.items(): print(key, "maps to", value)
```

```
1 maps to 42
2 maps to 17
3 maps to cats
dog maps to no cats
list maps to [1, 2]
```

```
[155]: d
```

```
[155]: {1: 42, 2: 17, 3: 'cats', 'dog': 'no cats', 'list': [1, 2]}
```

```
[156]: 2 in d
```

```
[156]: True
```

```
[157]: 4 in d
```

```
[157]: False
```

```
[158]: d[2]
```

```
[158]: 17
```

```
[159]: d[4]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[159], line 1
----> 1 d[4]

KeyError: 4
```

1.11 Exception Handling

```
[160]: try:
           print(d["dog"])
       except KeyError:
           print("Not found!")
```

```
no cats
```

```
[161]: try:
           print(d[42])
       except KeyError:
           print("Not found!")
```

```
Not found!
```

1.12 Comprehensions

Recall { x | 0 <= x < 5 } from math notation.

```
[162]: { x for x in range(0,5) }
```

```
[162]: {0, 1, 2, 3, 4}
```

```
[163]: {i * i for i in range(10)}
```

```
[163]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

```
[164]: [i*i for i in range(10)]
```

```
[164]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[165]: tuple(i*i for i in range(10))
```

```
[165]: (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

```
[166]: {i: i*i for i in range(10)}
```

```
[166]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Comprehension with filtering:

```
[167]: [i*i for i in range(10) if i % 2 == 0]
```

```
[167]: [0, 4, 16, 36, 64]
```

```
[168]: [0] * 10
```

```
[168]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
[169]: a = [[0]*10]*10
       a
```

```
[169]: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
[170]: a[0][0] = 42
       a
```

```
[170]: [[42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [42, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
[171]: a = [[0 for i in range(10)] for j in range(10)]
       a
```

```
[171]: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
[172]: a[0][0] = 42
       a
```

```
[172]: [[42, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

1.13 Generator objects

```
[173]: g = (i * i for i in range(10))
       g
```

```
[173]: <generator object <genexpr> at 0x10630cfb0>
```

```
[174]: type(g)
```

```
[174]: generator
```

```
[175]: for x in g: print(x)
```

```
0
1
4
9
16
25
36
49
64
81
```

[176]:
```
for x in g: print(x)
```

Generator items are consumed…

[177]:
```
g = (i * i for i in range(10))
```

[178]:
```
next(g)
```

[178]:
```
0
```

[179]:
```
next(g)
```

[179]:
```
1
```

[180]:
```
next(g)
```

[180]:
```
4
```

- A generator is special function that returns an stream whose items are consumed one at a time
- The contents of a genertor are not expanded in memory.

Range is also “lazy”:

[181]:
```
range(0,10)
```

[181]:
```
range(0, 10)
```

[182]:
```
type(range(0,10))
```

[182]:
```
range
```

But `range` is **not** a generator: - We cannot call `next` on range objects - A range is not consumed

1.13.1 A Simple Generator

[183]:
```
def simple_gen():
    yield 1
    yield 17
    yield 42
```

[184]:
```
g = simple_gen()
next(g)
```

[184]:
```
1
```

[185]:
```
next(g)
```

[185]:
```
17
```

[186]:
```
next(g)
```

[186]:
```
42
```

[187]:
```
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[187], line 1
----> 1 next(g)

StopIteration:
```

[188]:
```
g1 = simple_gen()
g2 = simple_gen()
print(next(g1))
print(next(g1))
print(next(g2))
```

```
1
17
1
```

[189]:
```
for i in simple_gen(): print(i)
```

```
1
17
42
```

[190]:
```
[x for x in simple_gen()]
```

[190]:
```
[1, 17, 42]
```

[191]:
```
list(simple_gen())
```

[191]:
```
[1, 17, 42]
```

1.13.2 A Fibonacci Generator

[192]:
```
def fib(n):
    """ This is the fibonacci function! """
    if n == 0: return 0
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a + b
    return b
```

[193]:
```
[fib(n) for n in range(10)]
```

[193]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

What is the problem..?

`Complexity is O(n^2)`

[194]:
```
def print_fib(n):
    a, b = 0, 1
    print(a)
    for i in range(n-1):
        print(b)
        a, b = b, a + b
```

[195]:
```
print_fib(10)
```

```
0
1
1
2
3
5
8
13
21
34
```

[196]:
```
def list_fib(n):
    l = []
    a, b = 0, 1
    l.append(a)
    for i in range(n-1):
        l.append(b)
        a, b = b, a + b
    return l
```

[197]:
```
list_fib(10)
```

[197]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Can we build a fibonacci generator?

[198]:
```
def gen_fib(n):
    a, b = 0, 1
    yield(a)
    for i in range(n-1):
        yield(b)
        a, b = b, a + b
```

[199]:
```
g = gen_fib(10)
```

[200]:
```
type(g)
```

[200]:
```
generator
```

[201]:
```
g1 = gen_fib(1000)
```

[202]:
```
print(g1)
```

```
<generator object gen_fib at 0x10638d380>
```

[203]:
```
for x in g: print(x)
```

```
0
1
1
2
3
5
8
13
21
34
```

[204]:
```
for x in g: print(x) # all elements are consumed
```

[205]:
```
def fib():
    a, b = 0, 1
    yield(a)
    while True:
        yield(b)
        a, b = b, a + b
```

[206]:
```
g = fib()
```

[207]:
```
type(g)
```

[207]:
```
generator
```

[208]:
```
# don't try this at home...
# for x in g: print(x)
```

[209]:
```
for i, x in enumerate([1,2,3]): print("the element", i, "has value", x)
```

```
the element 0 has value 1
the element 1 has value 2
the element 2 has value 3
```

[210]:
```
enumerate([1,2,3])
```

[210]:
```
<enumerate at 0x106387b00>
```

[211]:
```python
type(enumerate([1,2,3]))
```

[211]:
```
enumerate
```

`enumerate` can be used with any iterable item and adds a counter

[212]:
```python
g = fib()
for i, x in enumerate(g):
    print(x)
    if i >= 10: break
```

```
0
1
1
2
3
5
8
13
21
34
55
```

[213]:
```python
for i, x in enumerate(g):
    print(x)
    if i >= 5: break
```

```
89
144
233
377
610
987
```

[214]:
```python
next(g)
```

[214]:
```
1597
```

[215]:
```python
next(g)
```

[215]:
```
2584
```

[216]:
```python
g1 = fib()
```

[217]:
```python
next(g1)
```

[217]:
```
0
```

1.14 References

- Previous year's Jupyter notebooks
- https://docs.python.org/3/tutorial