Repetition

February 19, 2026

1 Repetition

1.1 Objects and variables

- All data in Python are objects
- Every object has a value, a data type and a unique ID
- Variables are names (identifiers) that refer to objects
- A variable can only refer to one object at a a time
- Assignment (=) makes a variable refer to a different object
- All data types have constructors, that attempt to coerce its argument to the data type

[1]:
```
int(5.0)
```

[1]:
```
5
```

[2]:
```
int('4')
```

[2]:
```
4
```

[3]:
```
try:
    int('four')
except ValueError:
    pass
```

[4]:
```
id(4)
```

[4]:
```
94821913484328
```

1.2 Data types

1.2.1 Simple data types - all immutable

- Numeric data types:
  - int - unlimited size
  - bool - an int in disguise 0 = False, everything else = True
  - float - IEEE 457
  - complex - two floats - real, imaginary
- Sequences:
  - str - Unicode code points - 1-4 bytes
  - bytes - 8 bits - often used for data import/export

1.2.2 Collection data types

- Mutable
  - list - sequence of objects
  - dict - collection of key -> value
  - set - a non ordered collection of uinque elements
- Immutable
  - tuple - a sequence of objects

1.3 Operators

1.3.1 Numeric types

- Numeric operators, i precedence order - (), **, ..., , / //, %, +, -, ..., =, +=, =

1.3.2 Sequence types

- + is concatenation
- * is repetition
- Indexing - indicies start at 0 - a[2]
- Slicing - [start] : [first element not included] (: step)
  - a[1:3] - element 1, 2
  - a[5:10:2] - element 5, 7, 9
  - a[10:4:-2] - element 10, 8, 6
  - a[3:] - element 3 to the end of the sequence
  - a[:5] - from the start to and including element 4
  - a[::-1] - reverse the sequence object
  - a[:] - copy the sequence object

2 Numeric types

[5]:
```
# Integers
i = 2
i += 2918340567787979789879
i
```

[5]:
```
2918340567787979789880 1
```

[6]:
```
# Floats
i += 1.0
i
```

[6]:
```
2.91834056778798e+22
```

[7]:
```
import math
math.sqrt(i)
```

[7]:
```
170831512543.4409
```

[8]:
```
# Complex
1.5 + 4j
```

[8]:
```
(1.5+4j)
```

[9]:
```
import cmath
cmath.sqrt(1.5+4j)
```

[9]:
```
(1.6988233976283063+1.177285409885548j)
```

3 Strings

[10]:
```
sentence = 'Gurkorna cyklar i motvind'
sentence[3]  # Indexing
```

[10]:
```
'k'
```

[11]:
```
sentence [9:15]  # Slicing
```

[11]:
```
'cyklar'
```

[12]:
```
sentence[:6]
```

[12]:
```
'Gurkor'
```

[13]:
```
sentence[-4:]
```

[13]:
```
'vind'
```

[14]:
```
sentence[::-1]
```

[14]:
```
'dnivtom i ralkyc anrokruG'
```

[15]:
```
sentence [::-2]
```

[15]:
```
'ditmirly norG'
```

[16]:
```
sentence.upper()
```

[16]:
```
'GURKORNA CYKLAR I MOTVIND'
```

[17]:
```
str(123.456)
```

[17]:
```
'123.456'
```

3.0.1 String escapes

Some characters have special meanings in regular strings. These are \, ' and ". If we want them to be interpreted as a regular character, we have to escape them, by preceding them with a \.

[18]:
```
print('abc\\\'\"\"')
```

```
abc\'""
```

3.0.2 Raw strings

Raw strings are preceded with the character *r*, and have no special meaning for the special characters in regular strings.

[19]:
```
print(r'\\"\"\u2103')
```

```
\\"\"\u2103
```

3.0.3 Format strings

Format strings are preceded by the character *f* and are the most recent way to put formatted data in strings.

[20]:
```
temperature = -4
sky = 'overcast'
print(f'The forecast for today is {sky} with a noon temperature of {temperature:
  ↪-6} \u2103.')
```

```
The forecast for today is overcast with a noon temperature of     -4 ℃.
```

http://fstring.help

4 Lists

[21]:
```
lucky_numbers = [5, 19, 47, -293]
dog_names = ['fluffy', 'fido', 'pluto', 'buster', 'kitty', 'fido']
senseless = [lucky_numbers, dog_names]
senseless
```

[21]:
```
[[5, 19, 47, -293], ['fluffy', 'fido', 'pluto', 'buster', 'kitty', 'fido']]
```

[22]:
```
lucky_numbers[2]  # Single object
```

[22]:
```
47
```

[23]:
```
lucky_numbers[2:3]  # List with 1 element
```

[23]:
```
[47]
```

[24]:
```
lucky_numbers[3] = -3  # The list is mutable
lucky_numbers
```

[24]:
```
[5, 19, 47, -3]
```

5 Tuples

[25]:
```
lucky_numbers2 = (5, 19, 47, -293)
dog_names2 = tuple(['fluffy', 'fido', 'pluto', 'buster', 'kitty', 'fido'])
senseless = (lucky_numbers2, dog_names2)
senseless
```

[25]:
```
((5, 19, 47, -293), ('fluffy', 'fido', 'pluto', 'buster', 'kitty', 'fido'))
```

[26]:
```
lucky_numbers2[2]  # Single object
```

[26]:
```
47
```

[27]:
```
lucky_numbers2[2:3]  # Tuple with 1 element
```

[27]:
```
(47,)
```

[28]:
```
try:
    lucky_numbers2[3] = -3  # The tuple is immutable
except TypeError:
    pass
lucky_numbers2
```

[28]:
```
(5, 19, 47, -293)
```

[29]:
```
lucky_numbers2 = lucky_numbers2[:-1] + (-3, )
lucky_numbers2
```

[29]:
```
(5, 19, 47, -3)
```

5.0.1 Unpacking

Sequence datatypes can be unpacked in assignment statements

[30]:
```
numbers = (1, 2, 3, 4)

first, second, third, fourth = numbers

print(fourth, second, third, first)
```

```
4 2 3 1
```

[31]:
```
numbers = [1, 2, 3, 4]

first, second, third, fourth = numbers
```

```
print(fourth, second, third, first, type(first))
```

```
4 2 3 1 <class 'int'>
```

[32]:
```
numbers = '1234'
first, second, third, fourth = numbers
print(fourth, second, third, first, type(first))
```

```
4 2 3 1 <class 'str'>
```

[1]:
```
x = [1, '2']
a, b = x
print(a, b, type(a), type(b))
```

```
1 2 <class 'int'> <class 'str'>
```

6 Dictionaries

[33]:
```
scores = {1: 'Lousy', 2: 'Almost ok', 3: 'Fine', 4: 'Good', 5: 'Fantastic'}
scores[2]
```

[33]:
```
'Almost ok'
```

[34]:
```
scores[0] = 'Abysmal'
scores
```

[34]:
```
{1: 'Lousy',
 2: 'Almost ok',
 3: 'Fine',
 4: 'Good',
 5: 'Fantastic',
 0: 'Abysmal'}
```

[35]:
```
scores.keys()
```

[35]:
```
dict_keys([1, 2, 3, 4, 5, 0])
```

[36]:
```
list(scores.keys())
```

[36]:
```
[1, 2, 3, 4, 5, 0]
```

[37]:
```
values = scores.values()
values
```

[37]:
```
dict_values(['Lousy', 'Almost ok', 'Fine', 'Good', 'Fantastic', 'Abysmal'])
```

[38]:
```
scores[5] = 'Superb'
values
```

[38]:
```
dict_values(['Lousy', 'Almost ok', 'Fine', 'Good', 'Superb', 'Abysmal'])
```

[39]:
```
reverse_scores = {}
for key, val in scores.items():
    reverse_scores[val] = key

reverse_scores
```

[39]:
```
{'Lousy': 1, 'Almost ok': 2, 'Fine': 3, 'Good': 4, 'Superb': 5, 'Abysmal': 0}
```

[40]:
```
reverse_scores['Abysmal']
```

[40]:
```
0
```

[41]:
```
reverse_scores2 = {val: key for key, val in scores.items()}
reverse_scores2
```

[41]:
```
{'Lousy': 1, 'Almost ok': 2, 'Fine': 3, 'Good': 4, 'Superb': 5, 'Abysmal': 0}
```

7 Sets

[42]:
```
lucky_numbers3 = {5, 19, 47, -293}
lucky_numbers3
```

[42]:
```
{-293, 5, 19, 47}
```

[43]:
```
list(lucky_numbers3)
```

[43]:
```
[19, 5, -293, 47]
```

[44]:
```
dog_names3 = set(['fluffy', 'fido', 'pluto', 'buster', 'kitty', 'fido'])
dog_names3
```

[44]:
```
{'buster', 'fido', 'fluffy', 'kitty', 'pluto'}
```

[45]:
```
cat_names = {'fluffy', 'kitty', 'macavity'}
dog_names3 & cat_names
```

[45]:
```
{'fluffy', 'kitty'}
```

[46]:
```
dog_names3 | cat_names
```

[46]:
```
{'buster', 'fido', 'fluffy', 'kitty', 'macavity', 'pluto'}
```

[47]:
```
dog_names3 - cat_names
```

[47]:
```
{'buster', 'fido', 'pluto'}
```

[48]:
```
cat_names - dog_names3
```

[48]:
```
{'macavity'}
```

[49]:
```
set('Gurkorna cyklar i motvind')
```

[49]:
```
{' ',
 'G',
 'a',
 'c',
 'd',
 'i',
 'k',
 'l',
 'm',
 'n',
 'o',
 'r',
 't',
 'u',
 'v',
 'y'}
```

8 Blocks in Python

In many other programming languages, blocks are denoted with BEGIN/END or { }. In Python, an extra indentation level denotes the start of a block, and going back to an earlier indentation level marks the end of one or more blocks. The recommended way of doing indentation is to use **4 spaces** for an indentation level, and **never to use <tab> characters** for block indentation. Most code editors translate the <tab> to spaces.

9 Conditional execution

Conditional execution is done with the *if* statement. Its general form is:

```
if <condition>:
    block
elif <condtion>:
    block
elif <condition>:
    block
else:
    block
```

Elif and *else* parts are optional.

9.0.1 Inline if

The inline *if* can be used inside expressions. It is fairly rarely used. Its general form is:

```
<expression> if <condition> else <expression>
```

[50]:
```
x = 4
z = 3 + (0 if x < 4 else 10)
z
```

[50]: 13

9.0.2 Truth values

The following evaluate to False: - False - None - Integer 0, Float 0.0, Complex 0+0j - Empty string (‘‘), empty bytes (b”) - Empty list, tuple, set or dict ([], (,), set([]), { }

All other values are True.

This allows for writing very short *conditions*.

[51]:
```
x = 0.0
bool(x)
```

[51]: False

[52]:
```
my_string = ''

if not my_string:
    print('The string is empty')
```

```
The string is empty
```

[ ]:
```

```

10 Match/Case

The *match/case* construct was introduced in Python 3.10. Until then, Python didn’t have an equivalent of switch/case in languages like C, C++ and Java. You could do multiple dispatch using dictionary lookup of functions to call, but that was not as readable as you would like it to be.

As usual, when Python introduced the functionality, it went over and beyond the implementation in other languages.

10.1 The simple form

Match/case will execute the first matching block only.

Note that you can use the reserved word *other* or *_* for the default case.

[53]:
```
designer = 'Bjarne'
```

```
match designer:
    case 'Larry':
        print(f'{designer} is the designer of Perl')
    case 'Dennis':
        print(f'{designer} is the designer of C')
    case 'Guido':
        print(f'{designer} is the designer of Python')
    case 'Brendan':
        print(f'{designer} is the designer of Javascript')
    case _:
        print(f'{designer} is not the designer of a notable programming␣
 ↪language')
```

```
Bjarne is not the designer of a notable programming language
```

10.2 Sequence matching

You can do matching on sequences. Literals have to be exactly matched, while variables get bound to whatever is in the matching position.

[54]:
```
sentence = 'Gurkorna cyklar i motvind'.split()

match sentence:
    case [a, b, 'i', 'motvind', 'nu']:
        print (a)
    case [a, b, 'i', 'motvind']:
        print (a, b)
    case [a, b, 'i', 'medvind']:
        print (a, b)
    case ['Gurkorna', 'cyklar', c, d]:
        print(c, d)
    case other:
        pass
```

```
Gurkorna cyklar
```

10.3 Alternatives and *args

You can use the | operator to indicate alternatives.

You can gather multiple positions in a single variable using a * in front of the variable, akin to how it is used in sequence unpacking.

[55]:
```
def file_handler(command):
    match command.split():
        case ['show']:
            print('List all files and directories: ')
            # code to list files
        case ['remove' | 'delete', *files] if '--ask' in files:
```

```
            del_files = [f for f in files if len(f.split('.'))>1]
            print('Please confirm: Removing files: {}'.format(del_files))
            # code to accept user input, then remove files
        case ['remove' | 'delete', *files]:
            print('Removing files: {}'.format(files))
            # code to remove files        case ['remove', *files]:

file_handler('show')
```

```
List all files and directories:
```

[56]:
```
file_handler('delete --ask valuable.c precious.py')
```

```
Please confirm: Removing files: ['valuable.c', 'precious.py']
```

[57]:
```
file_handler('delete whatever.pl')
```

```
Removing files: ['whatever.pl']
```

11 Assignment expressions

11.1 A K A The Walrus operator

There is new syntax *:=* that assigns values to variables as part of a larger expression. It is affectionately known as "the walrus operator" due to its resemblance to the eyes and tusks of a walrus.

Introduced in Python 3.8

In this example, the assignment expression helps avoid calling len() twice:

[58]:
```
a = [1] * 15
if (n := len(a)) > 10:
    print(f"List is too long ({n} elements, expected <= 10)")
n
```

```
List is too long (15 elements, expected <= 10)
```

[58]:
```
15
```

In regular expression matching where match objects are needed twice, once to test whether a match occurred and another to extract a subgroup:

[59]:
```
import re
advertisement = '18% discount'
discount = 0.0
if (mo := re.search(r'(\d+)% discount', advertisement)):
    discount = float(mo.group(1)) / 100.0
discount
```

[59]:
```
0.18
```

While-loops that compute a value to test loop termination and then need that same value again in the body of the loop:

[60]:
```
import io
f = io.StringIO('xyz' * 1000)
def process(x: str) -> None:
    pass

# Loop over fixed length blocks
while (block := f.read(256)) != '':
    process(block)

block = f.read()
if block:
    process()
```

Comprehensions where a value computed in a filtering condition is also needed in the expression body:

[61]:
```
from unicodedata import normalize

names = ['a', '\u2167', 'e\u0301','\u0061\u0301']
allowed_names = ['a', 'é', '\xe1', '\u2167']

[clean_name.title() for name in names
                    if (clean_name := normalize('NFC', name)) in allowed_names]
```

[61]:
```
['A', ' ', 'É', 'Á']
```

Try to limit use of the walrus operator to clean cases that reduce complexity and improve readability.

11.1.1 Sidenote on Unicode

Multiple ways to normalize

[62]:
```
normalize('NFKC', '\u2167')
```

[62]:
```
'VIII'
```

12 The *in* operator

A wonderful operator that checks if a value is in a *str*, *list*, *tuple*, *set*, *dict.keys()* or *dict.values()*.

[63]:
```
'buster' in dog_names
```

[63]:
```
True
```

[64]:
```
if 'buster' in cat_names:
    print('Buster is a cat name')
```

```
elif 'buster' in dog_names:
    print('Buster is a dog name')
else:
    print('Buster is neither a dog, nor a cat name')
```

```
Buster is a dog name
```

13 Iteration

13.0.1 For loops

A for loop has the general form:

```
for <variable> in <iterable>:
    block
else:
    block
```

The *else* is optional.

The iterable is an object that implements the *iteration* protocol. Sequence objects, like *str*, *list*, *tuple* and *dict.keys()*/*dict.values()*/*dict.items()* do.

13.0.2 While loops

While loops have the general form:

```
while <condition>:
    block
else:
    block
```

The *else* is optional.

13.0.3 Break statements

A *break* statement exits the loop.

13.0.4 Continue statements

A *continue* statement ends the current round in the loop and starts the next one at the top of the loop.

13.0.5 Else in for and while

The *else* part is executed if the loop was not exited with a *break*.

[65]:
```
for name in cat_names:
    print(name)
```

```
fluffy
kitty
macavity
```

[66]:
```
product = 2
while product < 100:
    print(product)
    product = product * product
```

```
2
4
16
```

[67]:
```
for name in cat_names:
    if name == 'macavity':
        break
    print(name)
else:
    print('Else reached')
```

```
fluffy
kitty
```

[68]:
```
for name in cat_names:
    if name == 'macavity':
        continue
    print(name)
else:
    print('Else reached')
```

```
fluffy
kitty
Else reached
```

13.0.6 The range function

To get sequences of integers in *for* loops and other places, Python uses the built in function *range*. It can be called with 1, 2 or 3 arguments.

```
range(<stop>)  # Start defaults to 0, step to 1
range(<start>, <stop>)  # Step defaults to 1
range(<start>, <stop>, <step>)
```

If *stop* is smaller than, or equal to, *start* and *step* is positive, no numbers will be produced. If *start* is smaller than, or equal to, *stop* and *step* is negative, no numbers will be produced.

[69]:
```
for i in range(3):
    print(i)
```

```
0
1
```

```
2
```

[70]:
```
for i in range(1, 3):
    print(i)
```

```
1
2
```

[71]:
```
for i in range(1, 5, 3):
    print(i)
```

```
1
4
```

[72]:
```
for i in range(100, 94, -2):
    print(i)
```

```
100
98
96
```

14 Comprehensions and generator statements

The various forms of comprehensions are a compact and readable way of manipulating collections. They take the general form:

`result = <startsymbol><expression> for <variable> in <iterable> if <expression><endsymbol>`

The if part is optional. The *startsymbol* and *endsymbol* are pairs of [], {}, (), depending on the variant we are using.

14.0.1 List comprehensions

Produce a list. Surrounded with [ ].

[73]:
```
squares = [x * x for x in range(11) if x % 3]
squares
```

[73]:
```
[1, 4, 16, 25, 49, 64, 100]
```

14.0.2 Set comprehensions

Produce a set. Surrounded with { }.

[74]:
```
squares = {x * x for x in range(11) if x % 3}
squares
```

[74]:
```
{1, 4, 16, 25, 49, 64, 100}
```

14.0.3 Dict comprehensions

Produce a dict. Surrounded with { }. *expression* takes the form

```
<key expression>: <value expression>
```

[75]:
```
squares = {str(x): x * x for x in range(11)}
squares
```

[75]:
```
{'0': 0,
 '1': 1,
 '2': 4,
 '3': 9,
 '4': 16,
 '5': 25,
 '6': 36,
 '7': 49,
 '8': 64,
 '9': 81,
 '10': 100}
```

14.0.4 Generator expressions

All the comprehensions produce a collection of data. This is impractical if

- you have a very large data set. You might run out of memory.
- you have a large data set (or an infinite iterator), and you only plan to use some of the data. You might waste a lot of CPU time.

A *generator expression* produces a small code object, that you can repeatedly ask for the next value, using the bult in function *next()*. Each time it returns one value. Please note that when you ask for a value, it gets consumed from the generator. The *for* statement has built in functionality for asking for each value, until the generator is exhausted.

[76]:
```
squares = (x * x for x in range(11))
squares
```

[76]:
```
<generator object <genexpr> at 0x7f4fc7adfac0>
```

[77]:
```
print(next(squares))
print(next(squares))
```

```
0
1
```

[78]:
```
for n in squares:
    print(n)
```

```
4
9
16
```

```
25
36
49
64
81
100
```

[79]:
```
try:
    next(squares)
except StopIteration:
    pass
```

15 Functions

Functions are a unit of reusablity in almost all programming languages. Python is no exception.

Python functions always return a value. If there is no explicit return value, the function returns the value *None*.

None is its own data type, which only contains the value *None*.

15.0.1 Function definition

A function is definined using the *def* statement. Optionally, we can define *parameters* for the function.

This is the shortest function definition we can make. This creates a *callable* object and binds it to the name *my_fun*. The *callable* contains a *code* object and other supporting objects, like the *parameter list* and *default values*.

[80]:
```
def my_fun():
    pass  # Does nothing
```

15.0.2 Function call

A function is called by giving the bound name, followed by parenthesis. Optionally, we can pass *arguments* to the call inside the parenthesis.

[81]:
```
print(my_fun())
```

```
None
```

15.0.3 Functions as first class objects

We can bind the function object to any variable name. We can also pass them as function arguments (example below).

[82]:
```
your_fun = my_fun
print(your_fun())
```

```
None
```

15.0.4 Return values

A function can return a value, using the *return* statement.

[83]:
```
from datetime import date
def calculate_now():
    return date.today()

print(calculate_now())
```

```
2025-09-23
```

15.0.5 Parameters

The regular parameters to a function are listed with parameter names.

[84]:
```
def multiply_and_add(multiplier1, multiplier2, add_factor):
    return multiplier1 * multiplier2 + add_factor
```

15.0.6 Arguments

We can call a function with *positional arguments*, *keyword arguments* or a combination. Positional arguments must come before keyword ones. Keyword arguments use the *parameter* name followed by = and a value.

[85]:
```
print(multiply_and_add(2, 3, 4))
print(multiply_and_add(2, 3, add_factor=4))
print(multiply_and_add(multiplier1=2, add_factor=4, multiplier2=3))
```

```
10
10
10
```

15.0.7 Default values

Parameters can have default values. This means that the default value is used if the argument is omitted. Parameters without default values should always come before parameters with them.

[86]:
```
def multiply_and_add(multiplier1, multiplier2=5, add_factor=3):
    return multiplier1 * multiplier2 + add_factor

print(multiply_and_add(2))
print(multiply_and_add(2, add_factor=4))
print(multiply_and_add(multiplier1=2, add_factor=4))
```

```
13
14
14
```

15.0.8 Never use mutable defaults

[87]:
```
def myfun(mylist=[]):
    mylist.append(2)
    print(mylist)

myfun()
myfun()
```

```
[2]
[2, 2]
```

15.0.9 Do this instead

[88]:
```
def myfun2(mylist=None):
    if not mylist:
        mylist = [2]
    print(mylist)

myfun2()
myfun2()
```

```
[2]
[2]
```

15.0.10 Catching extra positional arguments

A function can be called with a variable number of positional arguments if it has a paramter that is preceded by a *. This parameter must come after all regular parameters. It gathers all extra positional arguments in a *tuple*.

[89]:
```
def my_fun(a, b, *c):
    print(a, b, c)

my_fun(1, 2, 3, 4, 5)
```

```
1 2 (3, 4, 5)
```

15.0.11 Catching extra keyword arguments

A function can be called with a variable number of positional arguments if it has a paramter that is preceded by **. This parameter must come after all other parameters, including the one catching positional arguments. It gathers all extra keyword arguments in a dict.

[90]:
```
def my_fun(a, b, **c):
    print(a, b, c)

my_fun(1, b=2, x=3, biggest=4, smallest=5)
```

```
1 2 {'x': 3, 'biggest': 4, 'smallest': 5}
```

15.0.12 Unpacking a list/tuple in a function call

Sometimes you pass around collections that you want to use as arguments in a function call. Instead of doing tedious manual unpacking, you can get the collection unpacked directly in the call, by preceding the variable with a *.

[91]:
```
values = [1, 2, 3]

def my_fun(a, b, *c):
    print(a, b, c)

my_fun(*values)
# myfun(1,2,3)
```

```
1 2 (3,)
```

15.0.13 Unpacking a dict in a function call

In the same way, you can unpack a dictionary to become keyword arguments, by preceding it with **.

[92]:
```
values = {'b': 1, 'a': 2, 'x': 3}

def my_fun(a, b, **c):
    print(a, b, c)

my_fun(**values)
my_fun(b=1, a=2, x=3)
```

```
2 1 {'x': 3}
2 1 {'x': 3}
```

15.0.14 Passing on all arguments

We can pass on all arguments to a function without even knowing what they are.

[93]:
```
def fun1(*args, **kwargs):
    print('In fun1')
    fun2(*args, **kwargs)

def fun2(*args, **kwargs):
    print('In fun2')
    print(args, kwargs)

fun1(1, 2, x=4, y=[1, 2])
```

```
In fun1
In fun2
(1, 2) {'x': 4, 'y': [1, 2]}
```

15.0.15 Function as argument with lambda

[94]:
```
x = lambda n: float(n[1])
costs = [('b', '11'), ('a', '10.3'), ('x', '-2')]
print(sorted(costs, key=x))
sorted(costs, key=lambda n: float(n[1]))
```

```
[('x', '-2'), ('a', '10.3'), ('b', '11')]
```

[94]:
```
[('x', '-2'), ('a', '10.3'), ('b', '11')]
```

15.0.16 Function as argument without lambda

[95]:
```
def mysort(x):
    return (float(x[1]))

sorted(costs, key=mysort)
```

[95]:
```
[('x', '-2'), ('a', '10.3'), ('b', '11')]
```

15.0.17 A slightly more complex example

[96]:
```
from math import sin, cos, tanh, pi

nisse = sin
trig_functions = (sin, cos, tanh, nisse)

for fun in trig_functions:
    print(fun(pi/3))
```

```
0.8660254037844386
0.5000000000000001
0.7807144353592677
0.8660254037844386
```

15.0.18 Inner functions

Inner functions, also called *nested functions* are defined inside another function. They are only reachable form the the function enclosing them.

15.0.19 Scope

Variables defined in functions are local and can't be reached from enclosing scopes.

Variables in enclosing scopes can be accessed for reading if they are not hidden by being defined in the local scope or a closer enclosing scope.

Variables at the module global scope can be accessed for writing in the local scope by declaring them as *global*.

Variables from enclosing scopes (searched for in successive enclosing scopes excluding the global scope) can be accessed by declaring them as *enclosing*.

16 What is an object?

A set of data, and operations on those data.

17 What is a class?

A template for creating objects of a type - user defined type

[97]:
```
class Vehicle:
    pass

class Tank(Vehicle):
    pass

x = Tank()
isinstance(x, Tank)
```

[97]:
```
True
```

[98]:
```
isinstance(x, Vehicle)
```

[98]:
```
True
```

18 Skipped subjects

18.1 Functions

- Asyncronous functions
- Generator functions

18.2 Object oriented programming

- Multiple inheritance
- Class methods
- Static methods

[99]:
```
import datetime
now = datetime.datetime.now()
print(type(now))
print(now.strftime('%Y-%B-%d'))
```

```
<class 'datetime.datetime'>
2025-September-23
```

19 Decorators

[100]:
```
import time
from functools import wraps
def timeit(target_function):
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        before = time.time()
        result = target_function(*args, **kwargs)
        after = time.time()
        print(after-before)
        return result
    return wrapper

@timeit
def myfun():
    time.sleep(3)

# myfun = timeit(myfun)

myfun()
```

```
3.0009822845458984
```

[101]:
```
myfun.__name__
```

[101]:
```
'myfun'
```

20 Creating your own exception

All exceptions belong to a hierachy of classes. They behave just like any other class. Normally, you make your own exceptions by interiting from the *Exception* class, but sometimes you want to specialize one of the existing exceptions, like in the example below. In a try/except, an exception will catch not only itself, but also all of its subclasses.

[102]:
```
class MyException(ValueError):
    pass

try:
    pass
except MyException as e:
    ...
```

21 File processing

You always want to use a *context handler* when opening a file. This will ensure that the file gets closed, when you leave the context manager, even if an exception made that happen.

This example reads all the lines in a file, strips ending whitespace from each line (including the newline character). It then removes all empty lines and all comment lines.

[103]:
```
with open('simple.txt') as fp:
    lines = fp.readlines()
    lines = [line.rstrip() for line in lines]
    lines = [line for line in lines if line and not line.startswith('#')]
lines
```

[103]:
```
['Line 1', 'Line 2', 'Line 3', 'Line 4']
```

22 Regular expressions

This is a brief example of how regular expressions can be used.

[104]:
```
import re
pattern = r'^.*2019-1[01].*gnome.+'
with open('/var/log/alternatives.log') as logfile:
    lines = logfile.readlines()
    lines = [line for line in lines if re.match(pattern, line)]

for line in lines:
    print(line)
```