Python Cheat Sheet

neperiana.github.io

Python

Python is a widely-used, **interpreted, object-oriented**, and **high-level** programming language with dynamic semantics, used for general-purpose programming. Its philosophy emphasizes **code readability** with its use of **significant indentation**. Python was created as a hobby programming project and it is named after the BBC series Monty Python's Flying Circus.

The core philosophy of the language is summarized by the document "**PEP** 20 (The Zen of Python)", which includes aphorisms such as:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.

Many Pythons

There are two main kinds: Python 2 and Python 3. These aren't compatible with each other. Python 2 is an older version. Its development has since been stalled. **Python 3** is the newer (current) version of the language.

In addition, there is more than one version of each. There are **canonical** or **reference** Pythons which are maintained by the people gathered around the **PSF** (Python Software Foundation), a community that aims to develop, improve, expand, and popularize Python and its environment. All Pythons coming from the PSF are written in the "C" language. This is why the PSF implementation is often referred to as **CPython**.

Another Python family member is **Cython**, which tries to fix python lack of efficiency by automatically translating Python code (clean and clear, but not too swift) into "C" code (complicated and talkative, but agile). Other versions are Jython, PyPy or RPython.

print() & input()

The **print()** function sends data to the console, while the **input()** function gets data from the console.

```
print(
    "String 1",
    "String 2"
    sep=" ",
    end="\n",
)
```

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
```

Literals - the data in itself

Literals are notations for representing some fixed values in code:

- **Strings** – "Hello World"
- **Integers** (or simply ints) – 42
- **Floating-point numbers** (or simply floats) – 3.1415
- **Booleans** are the two constant objects True and False
- Special literal that is used in Python: the **None** literal, used to represent the absence of a value.

Expressions and Operators

An **expression** is a combination of values, variables or calls to functions which evaluates to a value, e.g., 1+2.
Operators are special symbols which are able to operate on the values and perform operations, e.g., the * operator.
There is a hierarchy of priorities: **unary** + and - have the highest priority, then: ****,** then: ***, /,** and **%,** and then the lowest priority: binary **+** and **-.**
Subexpressions in parentheses are always calculated first. The exponentiation operator uses right-sided binding.

Comments

Comments can be used to leave additional information in code, they are omitted at runtime. In Python, a comment is a piece of text that begins with **#**. You should give self-commenting names to variables.

Variables

A **variable** is a named location reserved to store values in the memory. Python is a **dynamically-typed** language, which means you don't need to declare variables. A variable is created or initialized automatically when you assign a value to it for the first time.
You can also use **compound assignment operators** to modify values assigned to variables, e.g., var += 1, or var /= 5 * 2.

Conditional Statements

When you want to execute some code only if a certain condition is met, you can use a **conditional statement**:

```
if x > 15: # False
    print("x > 15")

elif x > 10: # False
    print("x > 10")

elif x > 5: # True
    print("x > 5")

else:
    print("else will not be executed")
```

Loops

The **while loop** executes aa set of statements as long as a specified condition is true, e.g.:

```
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

The **for loop** executes a set of statements many times; it's used to iterate over a sequence, e.g.:

```
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

You can use **break** to exit a loop and **continue** to skip the current iteration. Loops can also have an **else** clause, which executes after the loop finishes (unless terminated by break statement).

Logical & Bitwise Operators

Python supports the following logical operators: **and**, **or** and **not**.

There are also bitwise operators to manipulate single bits of data:

- **&** does a bitwise and
- **|** does a bitwise or
- **~** does a bitwise not
- **^** does a bitwise xor
- **>>** does a bitwise right shift
- **<<** does a bitwise left shift

Lists

The **list** is a type of data used to store multiple objects. It is an **ordered** and **mutable** collection of comma-separated items between square brackets, e.g.:

```
myList = [1, None, True, "I am a string", 256, 0]
```

Lists can be **indexed** and **updated**. Lists can be **nested**. List **elements** and lists can be **deleted**. Lists can be **iterated** through using the for loop.

The **len()** function may be used to check the list's length.

List Operators (i)

- You can use the **sort()** method to sort elements of a list, e.g.: `lst.sort()`
- **sorted**() functions **sorts a copy of the list** instead.
- You can reverse the list with **reverse()** method.
- Note that l2 = l1 does not make a copy of the l1 list, but makes the variables l1 and l2 **point to the same list in memory**.
- If you want to copy a list or part of the list, you can do it by performing **slicing**. You can use negative indices to perform slices or omit them.

```
sliceOne = myList[2: ]
sliceTwo = myList[ :2]
sliceThree = myList[-2: ]
```

List Operators (ii)

- You can delete slices using the **del** instruction:

```
myList = [1, 2, 3, 4, 5]
del myList[0:2]
print(myList) # outputs: [3, 4, 5]
```

- You can test if some items exist in a list or not using the keywords in and not in, e.g.:

```
"A" in myList
```

- **List comprehension** allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

```
[expression for element in list if conditional]
```

- You can use **nested lists** in Python to create matrices or n-dimensional lists.

Functions

A **function** is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code **reusable**, **better organized**, and **more readable**. Functions can have **parameters** and **return values**.

There are at least four basic types of functions in Python:

- **built-in functions** which are an integral part of Python (such as the print() function)
- the ones that come from **pre-installed modules**
- **user-defined functions** which are written by users for users
- the **lambda functions**

You can define your own function using the **def** keyword and the following syntax:

```
def yourFunction(optional parameters):
    # the body of the function
```

Function Parameters

You can pass arguments to a function using the following techniques:

- **positional argument** passing in which the order of arguments passed matters.
- **keyword argument** passing in which the order of arguments passed doesn't matter.
- a mix of both. It's important to remember that positional arguments mustn't follow keyword arguments.
- You can use the keyword argument passing technique to pre-define a value for a given argument:

```
def name(firstN, lastN="Smith"):
    print(firstN, lastN)
```

Function Result

You can use the **return** keyword to tell a function to return some value. The return statement **exits** the function. You can use a list as a function's argument or as a function result too.

Variable Scope

- A variable that exists outside a function has a scope inside the function body unless the function defines a variable of the same name.
- A variable that exists inside a function has a scope inside the function body, but not outside.
- You can use the **global** keyword followed by a variable name to make the variable's scope global.

Recursion

When a function calls itself, this situation is known as **recursion**. The function calls itself and contains a specified termination condition (i.e., the **base case** - a condition which doesn't tell the function to make any further calls to that function). You can use recursive functions to write clean, elegant code, and divide it into smaller, organized chunks. Remember, recursive calls consume a lot of memory, and can be inefficient.

Tuples

Tuples are ordered and **immutable** collections of data. They can be thought of as immutable lists. They are written in **round brackets ()**.
You cannot change tuple elements (you cannot append tuples, or modify, or remove tuple elements) but you can use **del** to delete the whole tuple.

Dictionaries

Dictionaries are unordered*, changeable (mutable), and **indexed** collections of data. (*In Python 3.6x dictionaries have become ordered by default)
Each dictionary is a set of **key : value** pairs:

```
myDictionary = {
    key1 : value1,
    key2 : value2,
    key3 : value3,
    }
```

You can access an item by making a reference to its key inside a pair of square brackets or by using the **get()** method.

You can also insert an item to a dictionary by using the **update()** method, and remove the last element by using the **popitem()** method.

If you want to loop through a dictionary's keys and values, you can use the **items()** method.

```
for key, value in polEngDict.items()
```

To check if a given key exists in a dictionary, you can use the **in** keyword. You can use the **del** keyword to remove a specific item or delete a dictionary. To remove all the dictionary's items, you need to use the **clear()** method. To copy a dictionary, use the **copy()** method.

Packages

A **package** is a collection of related modules; in the world of modules, a package plays a similar role to a folder/ directory in the world of files.

Modules

Modules are parts of coded that are broken away for easier development. Each module consists of entities, like functions, variables, constants, classes, and objects. Each module is identified by its name. All these modules, along with the built-in functions, form the **Python standard library**.

In order to use a module or some of its content, first you need to import it using the **import** keyword.

A **namespace** is a space in which names exist and don't conflict with each other (each name is unique). Python imports the contents of a module, i.e., all the names defined in the module become known, but they don't enter your code's namespace. You can refer to them in your code as **module.name**. You can also import the entities directly, e.g.: `from math import pi`

Using **import *** you can import all entities from the indicated module (careful with name conflicts!). You can use the **as** keyword to give any imported entity the name you like - this is called **aliasing**.

To inspect the entities contained in a module you can use: `dir(module)`

Math Module

- **Trigonometry** functions, like sin(x), cos(x), tan(x), asin(x), acos(x), atan(x).
- Mathematical **constants**, like pi o e.
- **Degree** conversion, like radians(x) or degrees(x)
- **Exponentiation** functions, like exp(x), log(x), log(x, b), pow(x, y).
- Other general-purpose functions, like ceil(x), floor(x), trunc(x), factorial(x) or hypot(x).

Random Module (i)

- **random()** produces a float number between (0,1).
- **seed()** sets the generator seed for reproducibility.

Random Module (ii)

- **randint**(left, right) produces a random integer in the range [left, right].
- **choice**(sequence)
- **sample**(sequence, elements_to_choose=1)

Platform Module

- **platform**() lets you access the underlying hardware, operating system, and interpreter version.
- **machine**() tells you the generic name of the processor which runs your OS.
- **processor**() function returns a string filled with the real processor name.
- **system**() returns the generic OS name and **version**() the OS version.
- **python_implementation**() eturns a string denoting the Python implementation, usually Cpython, and **python_version_tuple**() returns a three-element tuple with major, minor, patch level numbers.

Creating Modules

When you run a file directly, its **__name__** variable is set to __main__. That's why files usually includes:

```
if __name__ == "__main__":
```

Can be used to include module test statements.

The convention of preceding the variable's name with _ or __ (underscores) indicates **hidden variables** that may be read but not modified.

You can use the **sys** package to tell python where to look for a user-built module:

```
from sys import path

path.append('..\\modules')
```

Creating Packages

To **create a package**, you can create an adequate folder structure. The outer folder will be the package's root.

Packages require to be **initialised**. To do so, you must include a file named **__init__.py.** The content of the file is executed when any of the package's modules is imported. If you don't want any special initializations, you can leave the file empty.

You can then find entities as *root.folder.folder.entity*

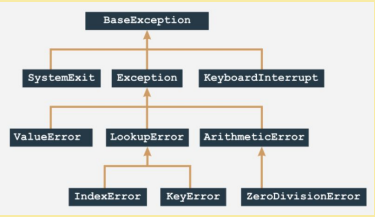
Exceptions (i)

Each time your code tries to do something wrong/foolish/irresponsible/crazy/unenforceable, Python does two things: it stops the execution and it raises an **exception**.

How do you **handle exceptions**? Use the **try** keyword, which introduces a block of the code which may or may not be performing correctly; the **except** keyword starts a piece of code which will be executed if anything inside the try block goes wrong.

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

Python 3 defines **63 built-in exceptions**, and all of them form a **tree-shaped hierarchy**, although the tree is a bit weird as its root is located on top.



Exceptions (ii)

To simulate raising an exception (to test your code or partially handle an exception somewhere else in your code) you can use the **raise** keyword:

```
raise exc
```

Assertions

The assert keyword evaluates an expression, if True, or a non-zero numerical value, or a non-empty string, or any other value different than None, it won't do anything else; otherwise, it automatically and immediately raises an exception named AssertionError.

Built-in Exceptions (i)

- **ArithmeticError**: an abstract exception including all exceptions caused by arithmetic operations like zero division or an argument's invalid domain. BaseException ← Exception ← …
- **AssertionError**: a concrete exception raised by the assert instruction when its argument evaluates to False, None, 0, or an empty string. BaseException ← Exception ← …
- **BaseException**: the most general (abstract) of all Python exceptions - all other exceptions are included in this one.
- **IndexError**: a concrete exception raised when you try to access a non-existent sequence's element (e.g., a list's element). BaseException ← Exception ← LookupError ← …
- **KeyboardInterrupt**: a concrete exception raised when the user uses a keyboard shortcut designed to terminate a program's execution (Ctrl-C in most OSs). BaseException ← …
- **LookupError**: an abstract exception including all exceptions caused by errors resulting from invalid references to different collections (lists, dictionaries, tuples, etc.) BaseException ← Exception ← …
- **MemoryError**: a concrete exception raised when an operation cannot be completed due to a lack of free memory. BaseException ← Exception ←…

Built-in Exceptions (ii)

- **OverflowError**: a concrete exception raised when an operation produces a number too big to be successfully stored. BaseException ← Exception ← ArithmeticError ← …
- **ImportError**: a concrete exception raised when an import operation fails. BaseException ← Exception ← StandardError ← …
- **KeyError**: a concrete exception raised when you try to access a collection's non-existent element (e.g., a dictionary's element). BaseException ← Exception ← LookupError ← …

Character Standards (i)

Computers store characters as numbers. They follow standards, like **ASCII** (short for American Standard Code for Information Interchange). ASCII is not enough for all languages though, The software **I18N** (internationalization) is a standard in present times. Each program has to be written in a way that enables it to be used all around the world, among different cultures, languages and alphabets.

A **code point** is a number which makes a character. We can say that standard ASCII code consists of 128 code points. A **code page** is a standard for using the upper 128 code points to store specific national characters. For example, there are different code pages for Western Europe and Eastern Europe, Cyrillic and Greek alphabets, Arabic and Hebrew languages, and so on.

Unicode assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points. The first 128 Unicode code points are identical to ASCII, and the first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page (a code page designed for western European languages). There is more than one standard describing the techniques used to implement Unicode in actual computers and computer storage systems. The most general of them is **UCS-4**, Universal Character Set.

Character Standards (ii)

UCS-4 uses 32 bits (four bytes) to store each character. As you can see, UCS-4 is a rather wasteful standard - it increases a text's size by four times compared to standard ASCII.

A more efficient encoding is **UTF-8**, stands for Unicode Transformation Format. UTF-8 uses as many bits for each of the code points as it really needs to represent them: all Latin characters (and all standard ASCII characters) occupy eight bits, non-Latin characters occupy 16 bits; CJK (China-Japan-Korea) ideographs occupy 24 bits.

Python 3 fully supports Unicode and UTF-8. This means that Python3 is completely I18Ned.

Strings (i)

- **Use backslash ** to escape character.
- Use **''' triple apostrophe** to extend string across lines.
- **\n** denotes a newline.
- **+** is used to **concatenate** strings
- ***** is used to **replicate** strings
- The function **ord()** (ordinal) takes a character and returns its ASCII/UNICODE code point value,.
- The function **chr()** takes a code point and returns its character.
- Strings aren't lists, but you can treat them like lists in many particular cases. You can **index, slice** strings or **iterate** through them. You can also use the **in** operator.
- **Strings are immutable**, you cannot use del keyword or append() function.
- Strings **can be compared** using the same set of operators (>,==,!=,..).
- The **index**() searches the sequence from the beginning, in order **to find the first element** of the value specified in its argument.
- The **list**() function takes its argument and **creates a new list containing all the string's characters**, one per list element.

Strings (ii)

- The **count**() method **counts all occurrences of the element** inside the sequence.
- The **capitalize**() method sets first character as upper case and remaining as lower case.
- The **center**() method makes a copy of the original string, trying to **center it inside a field of a specified width** using white spaces, unless another character is specified.
- The **endswith**() method checks if the given string ends with the specified argument. **startswith**() checks the start of the string.
- The **find**() method is similar to index(), it looks for a substring and returns the index of first occurrence of this substring, but **it's safer** as it doesn't generate an error for an argument containing a non-existent substring (it returns -1 then) and it works with strings only.
- The method **rfind**() does nearly the same things as find() but **start their searches from the end** of the string (hence the prefix r, for right).
- **isalnum**() checks if the string **contains only alphanumeric characters**. There are also **isalpha**() and **isdigit**() methods.
- The **islower**() method is a fussy variant of isalpha() - it accepts lower-case letters only. The **isupper**() method is the upper-case version of islower()
- The **isspace**() method identifies whitespaces only.
- The **lstrip**() method returns a newly created string formed from the original one by removing all leading whitespaces. If passed an extra parameter, it removes this characters from the original string. **rstrip**() does the same but starting from the end.
- The **trip**() method **combines the effects caused by rstrip() and lstrip()** - it makes a new string lacking all the leading and trailing whitespaces.
- The two-parameter **replace**() method returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument. The three-parameter replace() variant uses the third argument (a number) to limit the number of replacements.

Strings (iii)

- The **join**() method performs a join of the string list passed as argument, using the string from which the method has been invoked as a separator. Returns one string.
- The **split**() method **splits the string and builds a list of all detected substrings**. The method assumes that the substrings are delimited by whitespaces, but another character to split by can also be specified.
- The **lower**() method makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts.
- Similarly, the **upper**() method replaces all lower-case letters with their upper-case counterparts.
- The **swapcase**() method makes a new string by swapping the case of all letters within the source string.
- The **title**() method performs a somewhat similar function - it changes every word's first letter to upper-case, turning all other ones to lower-case.

Casting

- The **number-string conversion** is simple, as it is always possible. It's done by a function named **str**().
- The **string-number** transformation is possible when and only when the string represents a valid number. If the condition is not met, expect a ValueError exception. Use the **int**() function if you want to get an integer, and **float**() if you need a floating-point value.

OOP (i)

The **object approach** suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every class is like a recipe which can be used when you want to create a useful object (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.

Classes

A **class** is a model of a very specific part of reality, reflecting properties and activities found in the real world. The existence of a class does not mean that any of the compatible objects will automatically be created.

Functions embedded in classes are called **methods**. All methods must have at least one obligatory parameter, usually named **self** - it's only a convention, but you should follow it.

Python's convention to make an instance variable **protected** is to add a **single underscore (_)** to it. The **ability to actually hide or protect selected values or methods** against unauthorized access is called **encapsulation**. Python implements encapsulation when any class component has a name starting with **two underscores (_ _),** it becomes **private** - this means that it can be accessed only from within the class (unless accessed like **_className__propertyName**).

Objects

An **object** is a being **belonging to a class,** an incarnation of the requirements, traits, and qualities assigned to that specific class. The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an instance of the class).

The object programming convention assumes that every existing object may be equipped with three groups of attributes:

- a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
- a set of **individual properties** which make it original, unique or outstanding (although it's possible that some objects may have no properties at all).
- a **set of abilities** to perform specific activities, able to change the object itself, or some of the other objects.

Properties

There are two types of class properties:

- **Instance variables**: created inside methods. Modifying an instance variable of any object has no impact on all the remaining objects. They appear in the object __dict__.
- **Class variables**: defined outside of any method. They are not technically part of the object, so they do not appear in the object __dict__ but in the class __dict__. A class variable always presents the same value in all class instances.

To check an attribute existence on both objects and classes, the function **hasattr**(object/class, attributeName) can be used. Also note that we use the **dotted notation** when accessing an object property.

You can also use:

- **__dict__**property contains the names and values of all the properties the object/class is currently carrying.
- **__name__** to get the name of the class.
- **__module__** stores the name of the module which contains the definition of the class. __main__ if class is defined in file being run.
- **__bases__** is a tuple that contains classes (not class names) which are direct superclasses for the class. Only classes have this attribute - objects don't.

Methods (i)

A **method** is a function embedded inside a class. All methods must have at least one obligatory parameter, usually named **self**. The self parameter is used to obtain access to the object's instance and class variables as well as other class methods.

Each class needs a **constructor**, a function that construct a new object. The constructor should know everything about the object's structure and must perform all the needed initializations. The constructor's name is always **__init__.**

Methods (ii)

Note that constructors:

- **cannot return a value**, as it is designed to return a newly created object and nothing else.
- **cannot be invoked directly** either from the object or from inside the class.

When Python needs any class/object to be presented as a string (putting an object as an argument in the print() function invocation fits this condition) it tries to invoke a **method named __str__()** from the object and to use the string it returns.
The default __str__() method returns the previous string - ugly and not very informative. You can change it just by **defining your own __str__ method**.

Inheritance (i)

Classes have **hierarchy**. This hierarchy grows from top to bottom, like tree roots, not branches. The most general, and the widest, class is always at the top (the superclass) while its descendants are located below (the subclasses). Each **subclass is more specialized** (or more specific) than its superclass. Conversely, **each superclass is more general** (more abstract) than any of its subclasses.

This hierarchy implies inheritance. **Inheritance** is a common practice (in object programming) of passing attributes and methods from the **superclass** (defined and existing) to a newly created class, called the **subclass**. Any object bound to a specific level of a class hierarchy inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses.

To **extend classes** we define a new subclass pointing to the class which will be used as the superclass:

```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

Inheritance (ii)

Note that:

- The new subclass gets all the components defined by its superclass.
- Python forces you to explicitly **invoke a superclass's constructor**. It's generally a recommended practice to invoke the superclass's constructor before any other initializations you want to perform inside the subclass.

When we change the functionality of the methods of the extended class, but not their names we say that the push method has been **overridden**.

```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Useful:

- **issubclass**(ClassOne, ClassTwo) can check if a particular class is a subclass of any other class.
- **isinstance**(objectName, ClassName) tells you whether it is an object of a certain class or not.
- The **is** operator checks whether two variables (objectOne and objectTwo here) refer to the same object. Don't forget that variables don't store the objects themselves, but only the handles pointing to the internal Python memory.

Multiple inheritance occurs when a class has more than one superclass. But this is not best practice.

```
class Sub(SuperA, SuperB):
    pass
```

How Python **finds properties and methods**: When you try to access any object's entity, Python will try to (in this order):

- find it inside the object itself;
- in its superclasses, from bottom to top;
- if there is more than one class on a particular inheritance path, from left to right.

If all of the above fail, an exception (**AttributeError**) is raised.

Extending Classes (ii)

Note that:

- The new subclass gets all the components defined by its superclass.
- Python forces you to explicitly **invoke a superclass's constructor**. It's generally a recommended practice to invoke the superclass's constructor before any other initializations you want to perform inside the subclass.

When we change the functionality of the methods of the extended class, but not their names we say that the push method has been **overridden**.

When the subclass is able to override its superclass behaviour is called **polymorphism**.

The method, redefined in any of the superclasses, thus changing the behaviour of the superclass, is called **virtual**.

If the superclass method is empty because we are going to put all the details into the subclass, such a method is often called an **abstract method**, as it only demonstrates some possibility which will be instantiated later

Composition is the process of composing an object using other different objects.

We can say:

- inheritance extends a class's capabilities by adding new components and modifying existing ones
- composition projects a class as a container able to store and use other objects where each of the objects implements a part of a desired class's behaviour.

More Exceptions (i)

Try/except blocks can have:

- **else** branch that gets executed if no exception was raised
- **finally** branch that is always executed at the end

More Exceptions (ii)

Note that exceptions are a hierarchy of classes.

The BaseException class introduces a property named **args**. It's a tuple designed to gather all arguments passed to the class constructor.

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to **define your own, new exceptions** as subclasses derived from predefined ones.

Generators

A **generator** is a piece of specialized code able to produce a series of values, and to control the iteration process.

An object conforming to the **iterator protocol** is called an **iterator**. The iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the for and in statements. Must provide two methods:

- **__iter__()** which should return the object itself and which is invoked once
- **__next__()** which is intended to return the next value (first, second, and so on) of the desired series - it will be invoked by the for/in statements in order to pass through the next iteration; if there are no more values to provide, the method should raise the StopIteration exception.

When building a generator, you can replace return keyword in a normal function with the **yield keyword** and this **turns the function into a generator**.

```
def fun(n):
    for i in range(n):
        yield i
```

Such a function should not be invoked explicitly as - in fact - it isn't a function anymore; it's a generator object.

Generators and list comprehensions

Generators may also be used within **list comprehensions**.

The **list()** function can transform a generator invocation into a list.

Just **changing brackets [] to parentheses ()** can turn any comprehension into a generator.

Lambda functions

A **lambda function** is a function without a name, an anonymous function.

```
lambda parameters : expression
```

The **map()** function applies the function passed by its first argument to all its second argument's elements, and **returns an iterator** delivering all subsequent function results.

```
map(function, list)
```

The **filter()** function filters its second argument while being guided by directions flowing from the function specified as the first argument. The elements which return True from the function pass the filter - the others are rejected.

```
filter(funciton, list)
```

Closures

A **closure** is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore.

```
def outer(par):
    loc = par
    def inner():
        return loc
    return inner

var = 1
fun = outer(var)
print(fun())
```

In the example above, inner() is a closure.

Accessing Files

Note that there are **differences** in **file names** between Windows and Unix/Linux:

- systems derived from Unix/Linux don't use the disk drive letter (e.g., C:) and all the directories grow from one root directory called /, while Windows systems recognize the root directory as \
- two different separators for the directory names: \ in Windows, and / in Unix/Linux
- Unix/Linux system file names are case-sensitive. Windows systems store the case of letters used in the file name, but don't distinguish between cases

File Streams (i)

Note that there are **differences** in **file names** between Windows and Unix/Linux:

- systems derived from Unix/Linux don't use the disk drive letter (e.g., C:) and all the directories grow from one root directory called /, while Windows systems recognize the root directory as \
- two different separators for the directory names: \ in Windows, and / in Unix/Linux
- Unix/Linux system file names are case-sensitive. Windows files are not

Programs do not communicate with the files directly, but through some abstract entity called **handles** or **streams**. The operation of connecting the stream with a file is called **opening the file**, while disconnecting this link is named **closing the file**. The stream behaves like a tape recorder, a virtual head moves over the stream according to the number of bytes transferred from the stream.

There are two basic operations performed on the stream:

- **read** from the stream: the portions of the data are retrieved from the file and placed in a memory area managed by the program (e.g., a variable);
- **write** to the stream: the portions of the data from the memory are transferred to the file.

File Streams (ii)

The opening of the stream also declares the manner in which the stream will be processed, the open **mode**.

There are three basic modes used to open the stream:

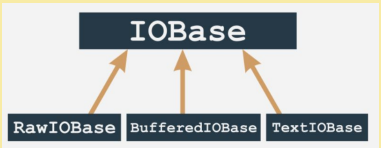
- **read mode**: a stream opened in this mode allows read operations only; trying to write to the stream will cause an exception (the exception is named UnsupportedOperation)
- **write mode:** a stream opened in this mode allows write operations only; attempting to read the stream will cause the exception mentioned above;
- **update mode:** a stream opened in this mode allows both writes and reads.

File Handles (i)

Every file is hidden behind an object of an adequate class, called file **handles**. An object of an adequate class is created when you open the file and annihilate it at the time of closing.

Note you never use constructors to bring these objects to life. The only way you obtain them is to invoke the function named open(). If you want to get rid of the object, you invoke the method named close().

In general, the object comes from one of the classes shown here:



Due to the type of the stream's contents, all the streams are divided into text and binary streams.

The **text streams** ones are structured in lines; they contain typographical characters arranged in rows (lines), as seen with the naked eye when you look at the contents of the file in the editor.

File Handles (ii)

This file is written (or read) mostly character by character, or line by line.

The **binary streams** don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio ... Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.

Then comes a subtle problem. In Unix/Linux systems, the line ends are marked by a single character \n. Other operating systems, like Windows, use a different convention: the end of line is marked by a pair of characters \r\n. The trait of the program allowing execution in different environments is called **portability**.

This issue is resolved at the level of classes, which are responsible for reading and writing characters to and from the stream. It works in the following way:

- when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is switched into text mode
- during reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a translation of newline characters occurs

Opening Streams (i)

The **opening of the stream** is performed by a function which can be invoked in the following way:

```
stream = open(file, mode = 'r', encoding = None)
```

The third parameter (encoding) specifies the encoding type (e.g., UTF-8 when working with text files).

Opening Streams (ii)

The second parameter (mode) can be one of:

- 'r' = **read mode**. The file associated with the stream must exist and has to be readable.
- 'w' = **write mode.** The file associated with the stream doesn't need to exist but it has to be writable; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased);
- 'a' = **append mode**. The file associated with the stream doesn't need to exist; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)
- 'r+' = **read and update mode**. the file associated with the stream must exist and has to be writeable. Both read and write operations are allowed for the stream.
- 'w+' = **write and update mode**. The file associated with the stream doesn't need to exist; if it doesn't exist, it will be created; the previous content of the file remains untouched; . Both read and write operations are allowed for the stream.

If there is a letter b at the end of the mode string it means that the stream is to be opened in the binary mode. If the mode string ends with a letter t the stream is opened in the text mode.

Text mode	Binary mode	Description
`rt`	`rb`	read
`wt`	`wb`	write
`at`	`ab`	append
`r+t`	`r+b`	read and update
`w+t`	`w+b`	write and update

Finally, the **successful opening** of the file will **set the current file position** (the virtual reading/writing head) **before the first byte of the file** if the mode is not a and after the last byte of file if the mode is set to a.

Opening Streams (iii)

Any stream operation must be preceded by the open() function invocation. There are three well-defined exceptions to the rule:

- sys.stdin, **standard input**
  - normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs; input() function reads data from stdin by default.
- sys.stdout, **standard output**
  - normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program; print() function outputs the data to the stdout stream.
- sys.stderr, **standard error output**
  - normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
  - the separation of stdout from the stderr gives the possibility of redirecting these two types of information to the different targets.

Closing Streams

The last operation performed on a stream should be closing.

```
stream.close()
```

Diagnosing Stream Problems (i)

The IOError object is equipped with a property named **errno**:

```
try:
    # some stream operations
except IOError as exc:
    print(exc.errno)
```

Diagnosing Stream Problems (ii)

Some common examples of errno are:

- **errno.EACCES → Permission denied**. The error occurs when you try, for example, to open a file with the read only attribute for writing.
- **errno.EBADF → Bad file number**. The error occurs when you try, for example, to operate with an unopened stream.
- **errno.EEXIST → File exists**. The error occurs when you try, for example, to rename a file with its previous name.
- **errno.EFBIG → File too large**. The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.
- **errno.EISDIR → Is a directory**. The error occurs when you try to treat a directory name as the name of an ordinary file.
- **errno.EMFILE → Too many open files**. The error occurs when you try to simultaneously open more streams than acceptable for your operating system.
- **errno.ENOENT → No such file or directory**. The error occurs when you try to access a non-existent file/directory.
- **errno.ENOSPC → No space left on device**. The error occurs when there is no free space on the media.

There is a function that can dramatically simplify the error handling code. Its name is **strerror(),** and it comes from the os module and expects just one argument - an error number.

```
except Exception as exc:
    print("The file could not be opened:", strerror(exc.errno));
```

I/O Streams (i)

Reading a text file's contents can be performed using several different methods:

- **read()** function: read a desired number of characters (or the whole file). When there is nothing more to read the function returns an empty string.

I/O Streams (ii)

- **readline()** method: read a complete line of text from the file, and returns it as a string in the case of success. Otherwise, it returns an empty string.
- **readlines()** method: when invoked without arguments, tries to read all the file contents, and returns a list of strings, one element per file line.
- **Iterating** through the object returned by the open() function (its __next__ method just returns the next line read in from the file.).

```
for line in open('text.txt', 'rt'):
    do_something(line)
```

Writing into a file can be done using:

- **write()** method: expects just one argument - a string that will be transferred to an open file.

Bytearrays

Amorphous data is data which have no specific shape or form - they are just a series of bytes. There are special containers able to handle such data - one of them is a specialized class name **bytearray** - as the name suggests, it's an array containing (amorphous) bytes.

```
data = bytearray(10)
```

Such an invocation creates a bytearray object able to store ten bytes. Note: such a constructor fills the whole array with zeros.

Bytearrays **resemble lists** in many respects. For example, they are mutable, they're a subject of the len() function, and you can access any of their elements using conventional indexing. There is one important limitation: they expect **integer values in the range 0 to 255** inclusive.

Function **hex()** can be used to print elements as hexadecimal values.

I/O Streams (iii)

Reading from a binary file requires use of a specialized method name **readinto(),** as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

```
bf = open('file.bin', 'rb')
bf.readinto(data)
bf.close()

for b in data:
    print(hex(b), end=' ')
```

An alternative way of reading the contents of a binary file is offered by the method named **read().** Invoked without arguments, it tries to **read all the contents of the file into the memory**, making them a part of a newly created object of the bytes class. Be careful - don't use this kind of read if you're not sure that the file's contents will fit the available memory.