DAY 1. Functional Programming

FUNCTIONAL PROGRAMMING:

https://www.geeksforgeeks.org/functional-programming-in-python/

- **Pure Functions:** These functions have two main properties. First, they always produce the same output for the same arguments irrespective of anything else. Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.
- **Recursion:** There are no “for” or “while” loop in functional languages. Iteration in functional languages is implemented through recursion.
- **Functions are First-Class and can be Higher-Order:** First-class functions are treated as first-class variables. The first-class variables can be passed to functions as a parameter, can be returned from functions or stored in data structures.
- **Variables are Immutable:** In functional programming, we can’t modify a variable after it’s been initialized. We can create new variables – but we can’t modify existing variables.

➔ Module, package - objects. Everything is an object.

➔ It’s not possible to declare **const var** in Python

Dunder methods (Double Under (Underscores))

https://www.geeksforgeeks.org/dunder-magic-methods-python/

Example: __call__, __init__, __repr__

Scopes Search in Python

Scopes: block and functional. Python has block scope.

LEGB - https://realpython.com/python-scope-legb-rule/

- **Local** (or function) scope is the code block or body of any Python function or `lambda` expression. This Python scope contains the names that you define inside the function. These names will only be visible from the code of the function. It's created at function call, *not* at function definition, so you'll have as many different local scopes as function calls. This is true even if you call the same function multiple times, or recursively. Each call will result in a new local scope being created.

  After function returns, GC (Garbage Collector) deletes everything from Local Execution Context

- **Enclosing** (or nonlocal) scope is a special scope that only exists for nested functions. If the local scope is an inner or nested function, then the enclosing scope is the scope of the outer or enclosing function. This scope contains the names that you define in the enclosing function. The names in the enclosing scope are visible from the code of the inner and enclosing functions.
- **Global** (or module) scope is the top-most scope in a Python program, script, or module. This Python scope contains all of the names that you define at the top level of a program or a module. Names in this Python scope are visible from everywhere in your code.
- **Built-in** scope is a special Python scope that's created or loaded whenever you run a script or open an interactive session. This scope contains names such as keywords, functions, exceptions, and other

attributes that are built into Python. Names in this Python scope are also available from everywhere in your code. It's automatically loaded by Python when you run a program or script. (Build-in functions: for, len, int etc.)

Scope identifiers in Python: **nonlocal, global**

➔ **ADDITIONAL CUSTOM MODULES COULD BE ADDED TO THE SYS PATH**

Module sys.py
> Import sys
> sys.path

➔ **Object is in standard library if should not be INSTALLED, if only imported its still in standard library**

Python Closure

https://www.geeksforgeeks.org/python-closures/

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

DESIGN PATTERNS:

- FACTORY,

- DECORATOR,
- ADAPTER,
- FASADE,
- FlyWeight

DECORATOR:
- Class decorator
- Function decorator
- Design pattern Decorator

➔ **Redeclaration and reinitialization is possible in Python**

```
print = lambda x: print('kot')
print('ela')
> 'kot'
```

➔ **Ternary operator python**

```
var = value1 if condition else value2
```

➔ **Difference between == and *is***
== - check value
is - check if it is exactly same object

➔ **Else in the loop**
while … else
for … else
else *always executes except when the loop has been broken (break)*

GENERATORS

- ALLOW TO CREATE COROUTINES
- **yield** - freeze the function (super return)

Difference between Iterators and Generators:
https://stackoverflow.com/questions/2776829/difference-between-pythons-generators-and-iterators
Generator is a subtype of iterator. An Iterator is an Iterable.
Iterator are objects which uses **next()** method to get next value of sequence.

A generator is a function that produces or yields a sequence of values using **yield** method.

Lazy evaluation:

```
>>> (x**2 for x in range(10))
<generator object <genexpr> at 0x104faad60>
>>> list((x**2 for x in range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Best Functional programming example:
map() - pure func, High order function, Lazy eval, lambda calculation

Hash function

What can be hashable ?
Immutable objects

Hash - string of characters. Always the same for the same value. It never repeats itself for different values.

```
>>> hash('Liza')
-8686124409037102302
>>> hash(-8686124409037102302)
-1768595381396020449
>>> hash(-1768595381396020449)
-1768595381396020449
>>>
```

Hash for int is always this int.

- ➔ A universally unique identifier (UUID) is **a 128-bit label used for information in computer systems**.

Functional programming - main points:

- Recursion over iteration (recursion is less effective)
- First class citizen
- Higher order function

- Lazy evaluation
- Pure function (without side effects)
- Closure
- Lambda calculation
- Currying: every function has only one parameter
- RHS - Right Hand Side, LHS - Left Hand Side
- Scope search: LEGB,
  L- local (function body)
  E - enclosure (function object)
  G - global (module)
  B - build - in (all that should not be installed)

DAY 2. Object Oriented Programming

Can we mix programming paradigms in Python? Probably yes, Python itself is created in different paradigms.

OOP: Encapsulation, Inheritance, Abstraction, Polymorphism

No private, protected, public identifiers in Python.

Class is a namespace in Python.
Every module in Python is a Singleton!
Can be imported only once, after changes can not be reimported!

Data Object Descriptor - protocol
Class @property
__get__
__set__
__delete__

Protocols

https://docs.python.org/3/library/collections.abc.html

```
# a + b -> __add__
# len(a) -> __len__
# a in [a, b, c] -> __contains__
# for state -> __iter__
# with statement -> __enter__, __exit__
# a == b -> __eq__
# f'' -> __format__

# sized -> __len__
# iterator -> __iter__, __next__
# context manager -> __enter__, __exit__
# data descriptor -> __get__, __set__, __delete__
# hashable -> __hash__
```

Tuple packing: a, b = 2, 5

Validation is necessary only when input is coming from a user!

No errors, but exceptions in Python!

Metaclasses vs. decorators

mc -> cls -> self
cls is instance of metaclass

Metaclass allows the create class factory.
However, decorators are advisable.
Decorators are less abstractive.
Multiple decorators can be used for 1 class, but only one metaclass can be a parent.

Default Metaclass for each class is a Type
Type is a Metaclass type( )

Encapsulation - hermetyzacja (izolowanie)

Access modifier:

- Private: __name (name mangling https://en.wikipedia.org/wiki/Name_mangling)
- Protected: _name
- Public: name
- Read-only: NAME

Class constructor __new__, not __init__

@classmethod works on a class, not on objects/ class instances.
Usage: multi type constructors

```
class A:
        myfield = 2

a = A( )
a.myfield = 4
print(A.myfield)  # still 2
```

DAY 3. Tests. TDD

There are **imperative** and **declarative** programming languages.

Reactive programming (Data Driven Development?)
Event Driven Development

TDD - Test Driven Development
BDD - Behaviour Driven Development

Tests:

➔ Evaluate the quality of a program
➔ After applying new changes check if everything works properly
➔ Detects errors earlier than in production

1. Unit test: only **atomic functionality** should be tested and only in **isolated environment**
2. Functional ← selenium ← driver ← browser
3. End to end (test whole business logic connecting few functionalities: buying flight ticket (login, fill date, pay money))
4. Integration (test how modules are integrated, connect all previous test types)
5. Regression tests
6. Performance tests
   Test API → postman, SoapUI → performance tests
7. Security tests → OWASP
8. Acceptance tests (product owner checks how requirements are implemented)

More about tests:

- test case, test suite, test plan
- assert
- smoke tests (test core functionalities)

Doublers

- db tests uses in-memory databases (Redis, H2)

Uses collaborator, substitute expensive functionalities

1. Stub (**simulate only needed class methods**, eliminate not needed and use cheaper implementation)
2. Fake (simulate all **class methods**, but in a cheaper simplified way)
3. Dummy (known or empty string)

Interaction

4. Spy (check if chosen method/function was tested and which arguments were given, uses **traceback**)
5. Mock (same as spy, but already know the this method was tested, so check the result not the fact of testing)

MonkeyPatch - replacing elements during runtime

Traceback - queue of program execution

Test case( ):
A arrange
A act
A assert

BDD → Selenium, Python, CSS
Also, web scraping, Robotic process automation (RPA)

TDD Concept:
Read → Green → Refactor

NoSQL - NOT ONLY SQL
databases are **schemaless**

PyTest
Fixture:

- set_up, tear_down before all tests
- set_up, tear_down before each test

TOX (run tests on different envs, simultaneously)
https://tox.readthedocs.io/en/latest/example/pytest.html

DAY 4. Algorithms & Data Structures

Python is interpreted and **just-in-time compilation**.

Composition in programming. Composition over iteration.

Stream - data in time

Microservices
Connection via HTTP, gRPC (google protocol)

WebSocket vs. HTTP

- HTTP (only request, response, no state. Slowly, because of connection establishment, certificate validation for each request)
- WebSocket is faster, listen all the time, accept requests and send responses via same connection simultaneously

Data Structures

Hash Map
Key can be the only immutable type (str, int, float, tuple etc.)

Linked List
Each element holds the reference to the next element

Int
Immutable**.** Limitless big number can be. Depends only on your PC.
Float
Immutable**.** Problem because 0.1 can be presented in the binary system.
Ex: 0.7 + 0.2

Str
Immutable. Set of chars.
Dict
Ordered from Python 3.7, ordered by keys in order of creation, however it is advisable to use OrderedDict.
Set
Unordered. Can be created only from hashable(immutable?) elements

Float → Decimal

Algorithms

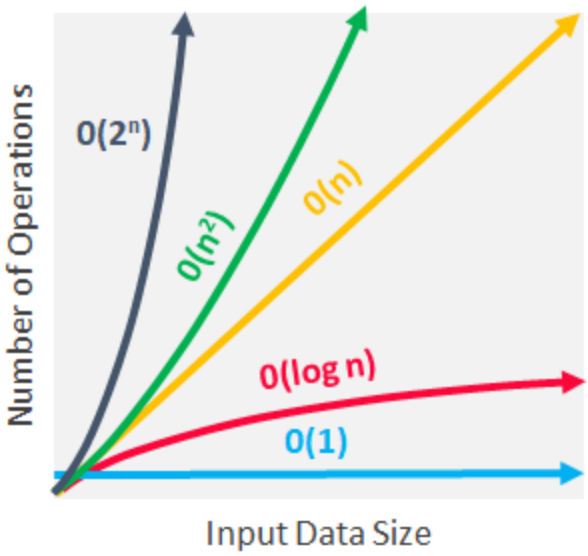
- Linear
- Branched (Conditional algorithms - Decision Tree)
- Recursive
- Iterative

Goals:
- Search
- Sort
- Add
- Edit
- …

"Divide and conquer"
"Greedy"
Dynamic programming

Computational Complexity

https://towardsdatascience.com/essential-programming-time-complexity-a95bb2608cac



1 - search by index, value assignment a = 2
Log n - search in sorted list
N - list iteration, search value in a list
N log n - sorting
n**2 - double for loop
2**n - ?

Space Complexity

https://towardsdatascience.com/space-and-time-complexity-in-computer-algorithms-a7fffe9e4683
https://www.geeksforgeeks.org/g-fact-86/

Design Pattern FlyWeight
https://www.geeksforgeeks.org/flyweight-method-python-design-patterns/

Graph

A graph is a type of non-linear data structure that is used to store data in the form of nodes and edges.

Distance between vertices
the minimum number of edges present between two nodes

Graph degree - max vertex degree of the graph
Binary tree is a graph with degree equals 2

Types of graphs

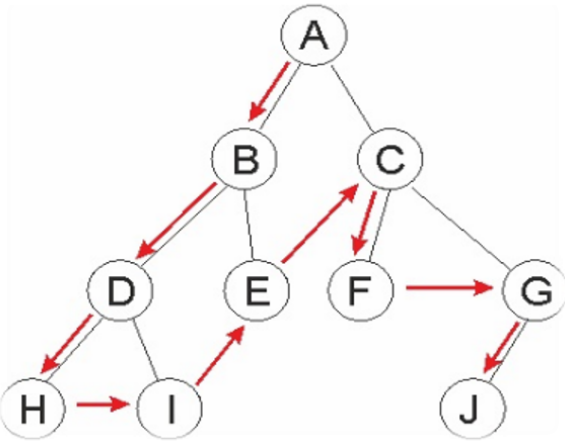
- ➔ Directed, Undirected
- ➔ Weighted and Unweighted
- ➔ Cyclic and Acyclic
- ➔ Connected and Disconnected
- ➔ Sparse and Dense

Heap

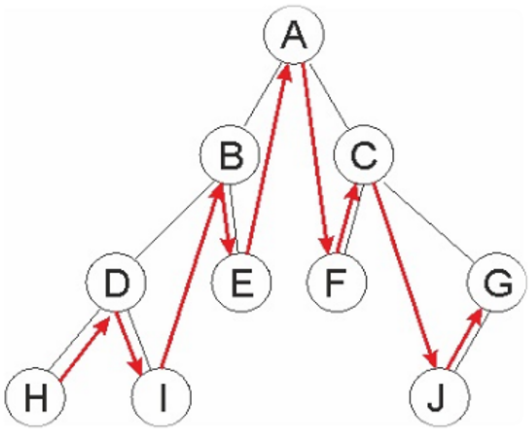
Heap is a special tree structure in which each parent node is less than or equal to its child node. Then it is called a **Min Heap**. If each parent node is greater than or equal to its child node then it is called a **Max heap**.

There are 3 ways of traversing the binary tree:
Longitudinal - **pre-order** (Polish notation):
Transverse - **in-order** (ordinary infix notation):
Reverse - **post-order** (reverse Polish notation):

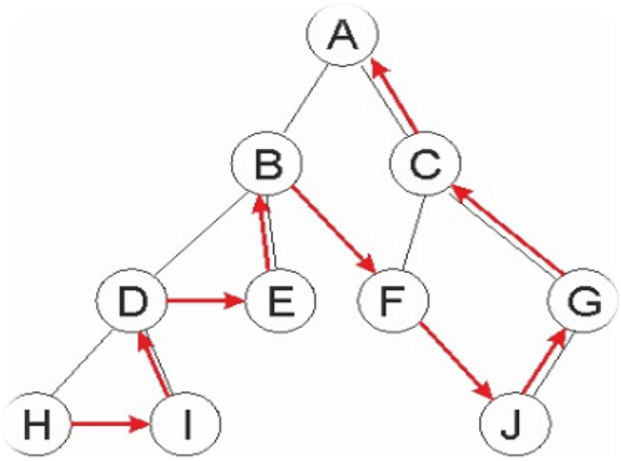
Pre-order
Root → left subtree → right subtree.



In-order
Left subtree → root → right subtree.



Post-order
Left subtree → right subtree → root.



DAY 5. Asynchronous programming

- **Function parameters** - a function is declared with certain parameters
- **Function arguments** - a function is called with passed arguments

def x (y=None) default parameters

def x (*args) positional parameters
def x (**kwargs) key parameters
def x (a, b, c, **/)** - parameters should be positional
def x **(***, r, t, v) - parameters should be key

AST - Abstract Syntax Tree, the process executed when the code is interpreted

__slots__ - reduces object size
https://www.geeksforgeeks.org/python-use-of-__slots__/#:~:text=slots%20provide%20a%20special%20mechanism,of%20memory%20optimisation%20on%20objects.&text=As%20every%20object%20in%20Python,dictionary%20that%20allows%20adding%20attributes.
__str__ → str( ), for end user
__repr__ → for developers, shows how object was created
eval( ) - run str as code
__format__ - accept modifiers for string, ex. %Y-%m

Asynchronous programming

Example:
A gas stove on which you can boil two pots of water "at the same time".

- Multithreading, Async (IO bound: requests to server or db etc.)
- Parallelism → Multiprocessing (CPU bound: math operations, machine learning etc.)

Multithreading - Threads:
- Native (depends on processor architecture)
- Virtual (depend on program)

GIL - Global Interpreter Lock (cPython)
Python has one main thread, multiple threads are locked.

Multiprocessing vs. Multithreading

1. Processes have their own **allocated memory**, however threads have **shared memory** that allows sending information between threads easier.
2. Each process runs on its own processor core and memory.

A **read–eval–print loop** (**REPL**), also termed an **interactive toplevel** or **language shell**

Jupyter - Julia, Python, R - data science main languages

MEMORY_PROFILER MODULE

To show how much memory and time is needed for code execution:

```
pip install memory_profiler

In [7]: %load_ext memory_profiler

In [8]: %memit (x ** 2 for x in range(100000))
peak memory: 44.97 MiB, increment: 0.20 MiB

In [9]: %memit [x ** 2 for x in range(100000)]
peak memory: 49.08 MiB, increment: 4.04 MiB

In [12]: %timeit (x ** 2 for x in range(100000))
592 ns ± 27.9 ns per loop (mean ± std. dev. of 7 runs, 1000000
loops each)

In [13]: %timeit [x ** 2 for x in range(100000)]
37.6 ms ± 1 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

Generators

- Pipeline → Generator Factory
- Package itertools
- **Coroutines are similar to generators, but we can add things to coroutines during execution, to generator CANNOT**

Future (Promise) - placeholder, object waiting for data, suspended object. Two states possible:

- Fulfilled
- Rejected

Callback - if something happens Do This

Synchronic is always before Async

- **GEC** - Global Execution Context
- Memory
- Callstack
- Queue
- **Event loop** is Subscriber

DESIGN PATTERN: Subscriber -> Subscription -> Observator

Task is considered as synchronic until the *await* keyword appears. So async code (coroutines) is only code with keyword *await* before.

Event loop checks:

1. Synchronous code to execute
2. Whether Promises/Futures are fulfilled
3. Whether GEC is not empty
4. Callbacks ???

GENERAL

Programming standards DRY, SOLID, PEP8

DRY - Don't repeat yourself

SOLID
S: Single Responsibility Principle (one responsibility for class, not multiple)
O: Open/Closed. Open for extension, closed for modification

L: Liskov substitution principle (every child can replace its parent, all functionalities)
I: Interface segregation (Interface explains logic of the class objects, but not implement it - separate logic and implementation)
D: Dependency inversion (Base class can not depend on child class)

PEP8

PyCHARM

1. TEMPLATES:

Preferences -> Live Templates
ds(skrot) + Tab

2. LOCAL HISTORY -> SHOW HISTORY
3. Shortcuts: Option+Commands+L - to clean code
4. Option+Enter - to generate class field assignment
5. Double Shift ! - search everywhere considering history
6. **Compare with Clipboard (local git diff)**

PROJECT FLOW

Product Owner creates high level requirements → Business Analyst creates business requirements → TECHLEAD analysis which techniques should be used → STORY → TASK

TASK
REQUIREMENTS
TEST SUITS
TEST CASE

Auto generated code: COPILOT, KITE