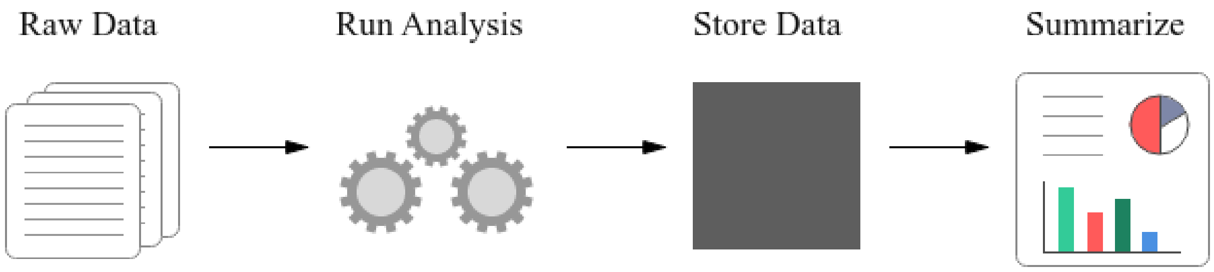
Functional Programming

Overview

In data engineering, the data pipeline is the real power behind production-grade databases, the optimization of SQL tables, and data structures and algorithms.

In a data pipeline, each task receives input and then returns output used in the next task.



The task is an abstract concept that we use to define parts of a pipeline. There is no right, or wrong way, to write a task. The only requirement is to allow inputs and return outputs that the next task can use as an input.

```
def task(input):
    output = do_something(input)
    return output
```

Understanding pure functions

When we only use functions, we call it *functional programming*. In functional programming functions are **stateless**, they rely *only on* their given inputs to produce an output.

Functions that meet the criteria for functional programming are called pure functions. Here's an example of the difference between pure and non-pure functions:

```
# Create a global variable `A`.
A = 5
def impure_sum(b):
    # Adds two numbers, but uses the
    # global `A` variable.
    return b + A
def pure_sum(a, b):
    # Adds two numbers, using
    # ONLY the local function inputs.
    return a + b
print(impure_sum(6))
print(pure_sum(6,3))
```

The benefit of using pure functions over non-pure functions is the reduction of side effects. Side effects occur when changes occur within a function's operation that are outside its scope. For example, they occur when we change the state of an object, perform any I/O operation, or even call `print()`:

```
def read_and_print(filename):
    with open(filename) as f:
        # Side effect of opening a
        # file outside of function.
        data = [line for line in f]
    for line in data:
        # Call out to the operating system
        # "println" method (side effect).
        print(line)
```

Programmers reduce side effects in their code to make it easier to follow, test, and debug. The more side effects a codebase has, the harder it is to step through a program and understand its sequence of execution.

While it's convenient to try and eliminate all side effects, they can often make programming easier. If we were to ban all side effects, then you wouldn't be able to read in a file, call print, or even assign a variable within a function. Advocates for functional programming understand this tradeoff, and they try to eliminate side effects where possible without sacrificing development implementation time.

<u>**The Lambda Expression**</u>

The `lambda` expression takes in comma-separated sequences of inputs (like `def`). Then, immediately following the colon, it returns the expression without using an explicit return statement. Finally, when assigning the `lambda` expression to a variable, it acts exactly like a Python function.

```
unsorted = [('b', 6), ('a', 10), ('d', 0), ('c', 4)]
# Sort on the second tuple value (the integer).
print(sorted(unsorted, key=lambda x: x[1]))

o/p = [('d', 0), ('c', 4), ('b', 6), ('a', 10)]
```

The Map function

The `map()` function takes in an iterable (i.e., `list`), and creates a new iterable object: a special `map` object. The first-class function applies to every element in the new object.

```
values = [1, 2, 3, 4, 5]

# Note: We convert the returned map object to
# a list data structure.
add_10 = list(map(lambda x: x + 10, values))
add_20 = list(map(lambda x: x + 20, values))

print(add_10)
```

It's important to cast the return value from `map()` as a `list` object.

The Filter function

The `filter()` function takes in an iterable, creates a new iterable object (again, a special `map` object), and a first-class function that must return a `bool` value. The new `map` object is a filtered iterable of all the elements that returned `True`.

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Note: We convert the returned filter object to
# a list data structure.
even = list(filter(lambda x: x % 2 == 0, values))
odd = list(filter(lambda x: x % 2 == 1, values))
print(even)
```

The Reduce Function

The `reduce()` function takes in a function and an iterable object such as a list. It will then reduce the list to a single value by successively applying the given function. It will first apply it on the first two elements and replace them with the result. Then it will apply the function on the first result and the next element and so on until a single value remains.

```
total_len = reduce(lambda x, y: len(x) + len(y) if isinstance(x, str) else x + len(y), ["I", "love", "data", "science"])
```

Writing Function Partials

The `partial` module takes in a function, and "freezes" any number of args (or kwargs), starting from the first argument, then returns a new function with the default inputs.

```
from functools import partial
def add(a, b):
    return a + b
add_two = partial(add, 2)
add_ten = partial(add, 10)

print(add_two(4))
o/p = 6
```

Using functional composition

if we've created a chain of function calls starting from `map` and ending with `reduce`. This chain of function calls has a term in mathematics called function composition. Given a chain of functions, `f(x), g(x), h(x)`, function composition occurs when you apply the output of each function to the input of the next: `h(g(f(x)))`.

This is exactly the same concept we used in our exercises:

reduce(filter(map(...)))

Using a function `compose` of a sequence of single argument functions, we can create a composed *single argument function* similar to the example above. Here's a composed function with `int` types instead of iterable types:

```
def add_two(x):
    return x + 2

def multiply_by_four(x):
    return x * 4

def subtract_seven(x):
    return x - 7

composed = compose(
    add_two,  # + 2
    multiply_by_four,  # * 4
    subtract_seven  # - 7
)

# (((10 + 2) * 4) - 7) = 41
answer = composed(10)
print(answer)
```

Ex-2

```
lines = read('example_log.txt')
ip_addresses = list(map(lambda x: x.split()[0], lines))
filtered_ips = list(filter(lambda x: int(x.split('.')[0]) <= 20, ip_addresses))

ratio = count_filtered / count_all
extract_ips = partial(
    map,
    lambda x: x.split()[0]
)
filter_ips = partial(
    filter,
    lambda x: int(x.split('.')[0]) <= 20
)
count = partial(
    reduce,
    lambda x, _: 2 if isinstance(x, str) else x + 1
)
composed = compose(
    extract_ips,
    filter_ips,
    count)
counted = composed(lines)
```

Pipeline Tasks

Generators in Python

In the previous lesson, we would read in the `example_log.txt` file, and write it to a list. Recall that when creating a list, Python loads each element of the list into RAM. For files that exceed multiple gigabytes, this file loading can cause a program to run out of memory.

Instead of reading the file into memory, we can take advantage of file streaming. **File streaming works by breaking a file into small sections (called chunks), and then loaded one at time into memory. Once a chunk has been exhausted (all the bytes of that chunk have been read), Python requests the next chunk, and then that chunk is loaded into memory to be iterated on.**

This stream-like behaviour is extremely helpful when working with large data sets. We can replicate this behaviour with other iterators with the use of generators. A **generator** is an *iterable object* that is created from a **generator** function.

The generator function differs from a regular function by two important differences:

- A generator uses `yield`, instead of `return`. (However, a `return` statement is used to stop iteration, more on that soon).
- Local variables are kept in memory until the generator completes.

The `yield` expression is responsible for two actions:

- A signal to the Python interpreter that this function will be a generator.
- Suspends the function execution, keeping the local variables in memory, until the next call.

The suspension of execution, saving local variables, and then resuming operation is what allows the generator to act like a stream.Using the `next()` function, you can see the generator and yield suspension work in action. In an iteration (like a `for` loop), the Python interpreter continuously calls the `next()` function to receive the "next" element in the iterable. In a generator, each call to the `next()` function completes a cycle, and then stops at the next `yield`.

Suppose we wanted to give an upper limit to the `count()` function. Then we need to use a `return` statement within the generator. The `return` statement is one way that a Python loop (eg. `for`) knows when to stop looping. Using `return` without an argument ends the function and returns `None`, breaking the loop.

Generators Comprehension

Generator comprehensions are extremely similar to list comprehensions. We can turn any list comprehension into a generator comprehension by replacing the square brackets `[]` to parenthesis `()`. For example, here's how we could write the `squares` function in the previous screen as a list and generator expression:

squared_list = [i * i for i in range(20)]

squared_gen = (i * i for i in range(20))

Before you begin replacing all your lists as generators, let's discuss a major drawback of the generator. Suppose we had two places in our code that wanted to use the `squared_gen` generator. With a list, `squared_list`, we could easily do:

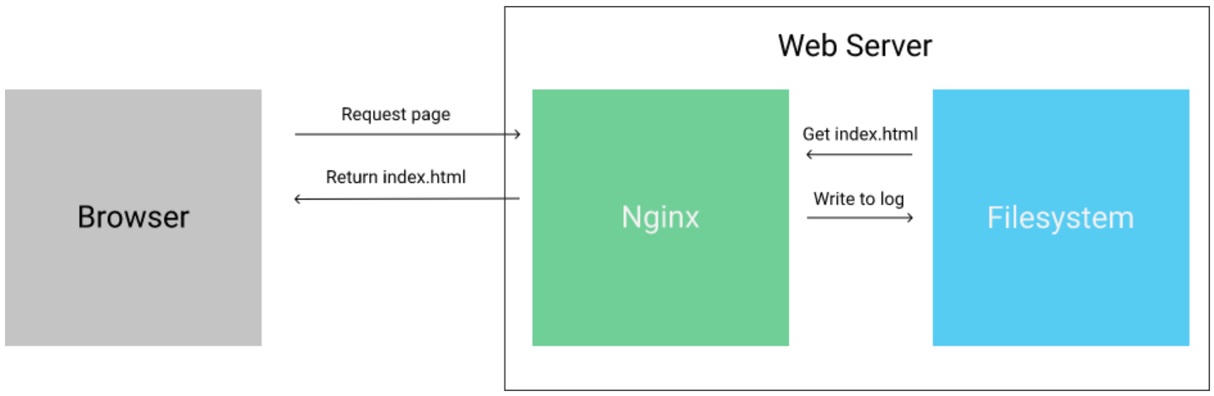
```
num_to_square = {}
for idx, i in enumerate(squared_list):
    num_to_square[idx] = i
print(num_to_square)
{0: 0, 1: 1, 2: 4, 3: 9 ...}
for i in squared_list:
    print(i)
0
1
4
9

...
```

Using a generator, however, the second loop will not run. Like a file, a generator will *exhaust* all its elements once the final yield has been executed. Be cautious of this behaviour when using generators, like `squared_gen` in your code!

```
num_to_square = {}
for idx, i in enumerate(squared_gen):
    num_to_square[idx] = i
print(num_to_square)
{0: 0, 1: 1, 2: 4, 3: 9 ...}
for i in squared_gen:
    print(i)
None
```

Manipulating Generators in Tasks

The goal of our pipeline will be to take log lines, from the `example_log.txt` file, and create a summary CSV file of unique HTTP request types and their associated counts. Each line from the `example_log.txt` file is from an NGINX log file. This log file contains each client request sent to a running web server in a client-server model.



Here's the first few lines of the log:

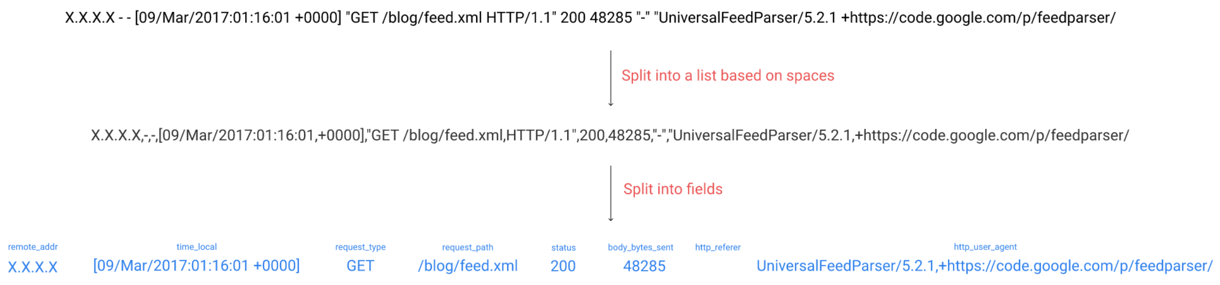
```
200.155.108.44 - - [30/Nov/2017:11:59:54 +0000] "PUT /categories/categories/categories HTTP/1.1" 401 963 "http://www.yates.com/list/tags/category/" "Mozilla/5.0 (Windows CE) AppleWebKit/5332 (KHTML, like Gecko) Chrome/13.0.864.0 Safari/5332"
```

```
$remote_addr - $remote_user [$time_local] "$request"
$status          $body_bytes_sent          "$http_referer"
"$http_user_agent"
```

- `$remote_addr` — the ip address of the client making the request to the server.
- `$remote_user` — if the client authenticated with basic authentication, this is the user name (blank in the examples above).
- `$time_local` — the local time when the request was made.

- `$request` — the type of request, and the URL that it was made to.
- `$status` — the response status code from the server.
- `$body_bytes_sent` — the number of bytes sent by the server to the client in the response body.
- `$http_referrer` — the page that the client was on before sending the current request.
- `$http_user_agent` — information about the browser and system of the client.
-

```
log = open('example_log.txt')
def parse_log(log):
    for line in log:
        split_line = line.split()
        remote_addr = split_line[0]
        time_local = split_line[3] + " " + split_line[4]
        request_type = split_line[5]
        request_path = split_line[6]
        status = split_line[8]
        body_bytes_sent = split_line[9]
        http_referrer = split_line[10]
        http_user_agent = " ".join(split_line[11:])
        yield (
            remote_addr, time_local, request_type, request_path,
            status, body_bytes_sent, http_referrer, http_user_agent
        )

first_line = next(parse_log(log))
```



```
X.X.X.X - - [09/Mar/2017:01:16:01 +0000] "GET /blog/feed.xml HTTP/1.1" 200 48285 "-" "UniversalFeedParser/5.2.1 +https://code.google.com/p/feedparser/
```

```
X.X.X.X,-,-,[09/Mar/2017:01:16:01,+0000],"GET /blog/feed.xml,HTTP/1.1",200,48285,"-","UniversalFeedParser/5.2.1,+https://code.google.com/p/feedparser/
```

remote_addr	time_local	request_type	request_path	status	body_bytes_sent	http_referer	http_user_agent
X.X.X.X	[09/Mar/2017:01:16:01 +0000]	GET	/blog/feed.xml	200	48285		UniversalFeedParser/5.2.1,+https://code.google.com/p/feedparser/

Data Cleaning in Parse Log

```
def parse_time(time_str):
    """
    Parses time in the format [30/Nov/2017:11:59:54 +0000]
    to a datetime object.
    """
    time_obj = datetime.strptime(time_str, '[%d/%b/%Y:%H:%M:%S %z]')
    return time_obj

def strip_quotes(s):
    return s.replace('"', '')

log = open('example_log.txt')

def parse_log(log):
    for line in log:
        split_line = line.split()
        remote_addr = split_line[0]
        time_local = parse_time(split_line[3] + " " + split_line[4])
        request_type = strip_quotes(split_line[5])
        request_path = split_line[6]
        status = int(split_line[8])
        body_bytes_sent = int(split_line[9])
        http_referrer = strip_quotes(split_line[10])
        http_user_agent = strip_quotes(" ".join(split_line[11:]))
        yield (
            remote_addr, time_local, request_type, request_path,
            status, body_bytes_sent, http_referrer, http_user_agent
        )


first_line = next(parse_log(log))
```

Write to CSV

Question:-

- Write a function `build_csv()` that takes in a required argument, `lines` (the parsed rows), `file` and optional argument `header`.
    - If there is a `header` argument, insert the header at the beginning of the parsed rows.
    - Write the CSV to the given `file`.
    - Return the `file`.
- Open the file `temporary.csv` for reading and writing.
- Call `build_csv()` with the following variables:
    - `parsed` for lines.

    - A `list` or `tuple` of the header names in the screen's example.
    - The `temporary.csv` file.
- Assign the `build_csv()` return value to the variable `csv_file`.
- Call `csv_file.readlines()` and assign the return value to the variable `contents`.
- Print the first 5 rows of `contents` using `print()`.

Answer :-

```
import csv

log = open('example_log.txt')
parsed = parse_log(log)

def build_csv(lines, file, header=None):
    if header:
        lines = [header] +[l for l in lines]
    writer = csv.writer(fil, delimiter=',')
    writer.writerows(lines)

    file.seek(0)
    return file

file = open('temporary.csv', 'r+')
csv_file = build_csv(parsed, file, header =[
        'ip', 'time_local', 'request_type',
        'request_path', 'status', 'bytes_sent',
        'http_referrer', 'http_user_agent'
    ])

contents = csv_file.readlines()
print(contents[:5]
```

Chaining Iterators

The `itertools.chain()` function combines a list of iterables together to create a single iterable object that runs through every element.

```
import csv

import itertools

log = open('example_log.txt')
parsed = parse_log(log)
```

```
def build_csv(lines, file, header=None):
    if header:
        lines = itertools.chain([header], lines)
    writer = csv.writer(file, delimiter=',')
    writer.writerows(lines)
    file.seek(0)
    return file

file = open('temporary.csv', 'r+')
csv_file = build_csv(
    parsed,
    file,
    header=[
        'ip', 'time_local', 'request_type',
        'request_path', 'status', 'bytes_sent',
        'http_referrer', 'http_user_agent'
    ]
)

contents = csv_file.readlines()
print(contents[:5])
```

Counting Unique Request Types

```
import csv
import itertools

def count_unique_request(csv_file):
    reader = csv.reader(csv_file)
    header = next(reader)
    idx = header.index('request_type')

    uniques = {}
    for line in reader:
        if not uniques.get(line[idx]):
            uniques[line[idx]] = 0
        uniques[line[idx]] += 1
    return uniques

log = open('example_log.txt')
parsed = parse_log(log)
file = open('temporary.csv', 'r+')
csv_file = build_csv(parsed,file,
    header=[
        'ip', 'time_local', 'request_type',
        'request_path', 'status', 'bytes_sent',
        'http_referrer', 'http_user_agent'
    ]
)
uniques = count_unique_request(csv_file)
print(uniques)
```

Task Reusability

```
import csv
def count_unique_request(csv_file):
    reader = csv.reader(csv_file)
    header = next(reader)
    idx = header.index('request_type')

    uniques = {}
    for line in reader:
        if not uniques.get(line[idx]):
            uniques[line[idx]] = 0
        uniques[line[idx]] += 1
    return ((k,v) for k,v in uniques.items())
log = open('example_log.txt')
parsed = parse_log(log)
file = open('temporary.csv', 'r+')
csv_file = build_csv(
    parsed,
    file,
    header=[
        'ip', 'time_local', 'request_type',
        'request_path', 'status', 'bytes_sent',
        'http_referrer', 'http_user_agent'
    ]
)
uniques = count_unique_request(csv_file)
summarized_file = open('summarized.csv', 'r+')
summarized_csv = build_csv(uniques, summarized_file , header = ['request_type', 'count'])
print(summarized_file.readlines())
```

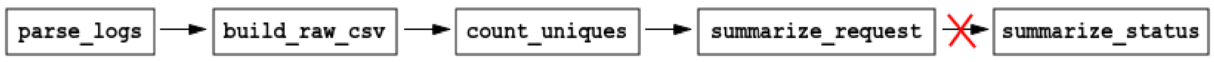
Building a Pipeline class

```
import io

class Pipeline:
    def __init__(self):
        self.tasks = []

    def task(self, depends_on=None):
        idx = 0
        if depends_on:
            idx = self.tasks.index(depends_on) + 1
        def inner(f):
            self.tasks.insert(idx, f)
            return f
        return inner

    def run(self, input_):
        output = input_
        for task in self.tasks:
            output = task(output)
        return output

pipeline = Pipeline()

@pipeline.task()
def parse_logs(logs):
    return parse_log(logs)

@pipeline.task(depends_on=parse_logs)
def build_raw_csv(lines):
    return build_csv(lines, header=[
        'ip', 'time_local', 'request_type',
        'request_path', 'status', 'bytes_sent',
        'http_referrer', 'http_user_agent'
    ],
    file=io.StringIO())

@pipeline.task(depends_on=build_raw_csv)
def count_uniques(csv_file):
    return count_unique_request(csv_file)

@pipeline.task(depends_on=count_uniques)
def summarize_csv(lines):
    return build_csv(lines, header=['request_type', 'count'], file=io.StringIO())

log = open('example_log.txt')
summarized_file = pipeline.run(log)
print(summarized_file.readlines())


# Check val
```

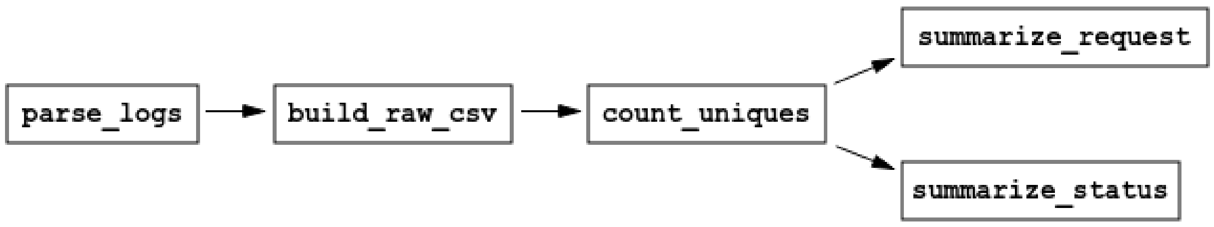
```
"['request_type,count\r\n', 'GET,3334\r\n', 'POST,3299\r\n', 'PUT,3367\r\n']"
```

Multiple dependency pipeline

In our last lesson's task pipeline, the final task was to summarise logs that are outputted from a parsed CSV file. The summary is run on the `request_type` column name, but suppose that we wanted to also run a summary on the `status` column. This seems doable – our only requirement should be the parsed CSV – but with our linear pipeline this will not work.



Instead of a linear ordering, we need the ability to create multiple branches of dependencies. We're looking to build a data structure that can support the following task pipeline:



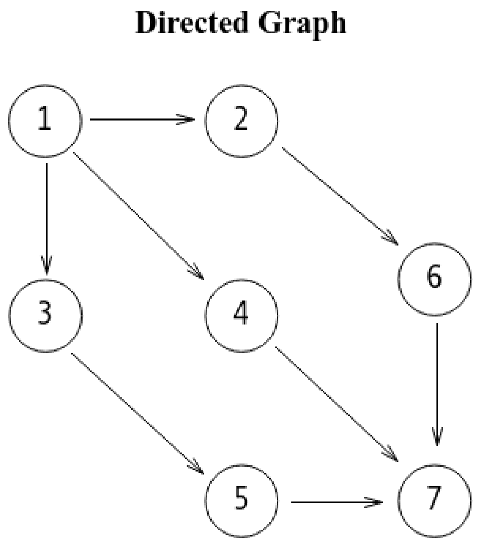
Intro to DAGs

In the introduction, we briefly mentioned the concept of a DAG. To describe the DAG, we're going to supplement the language of a graph with identical terms used in the course on trees. Let's break down what each of the terms mean:

- Graph: The data structure is composed of vertices (*nodes*) and edges (*branches*).
- Directed: Each *edge* of a *vertex* points only in one direction.
- Acyclic: The graph does not have any cycles.

The graph definition is ambiguous, like the definition of a tree, so it's easier to describe with diagrams.

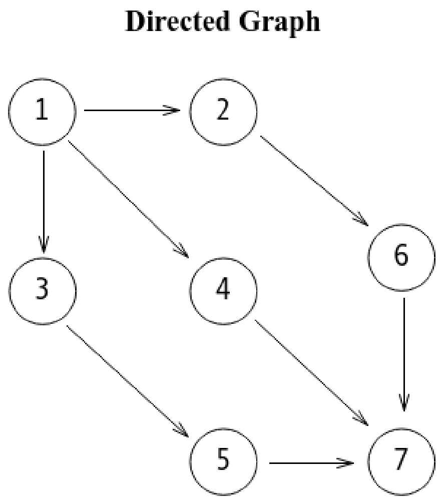
The vertices (or nodes) are each point in the graph, and the edges are the lines that connect them. Note that there is no requirement of the direction of each edge (ie. The edge (2, 1) and (1, 2) are both possible). If we wanted to restrict the direction of an edge to only one possible combination, then it would be called a *directed* graph.



The edge arrows correspond to the direction of the edges, which written out as a tuples, would be (1, 2), (2, 6), or (3, 5). If we follow a sequence of directed edges, like [(1, 3), (3, 5), (5, 7)], then we call this sequence a path of the graph.

If there are any paths that starts, and end with one vertex, then the graph contains a *cycle*. For example, if we had a path like [(1, 3), (3, 5), (5, 4), (4, 1)], then it would be a cycle. If a graph does not contain any cycles, then it is *acyclic*.

The DAG Class

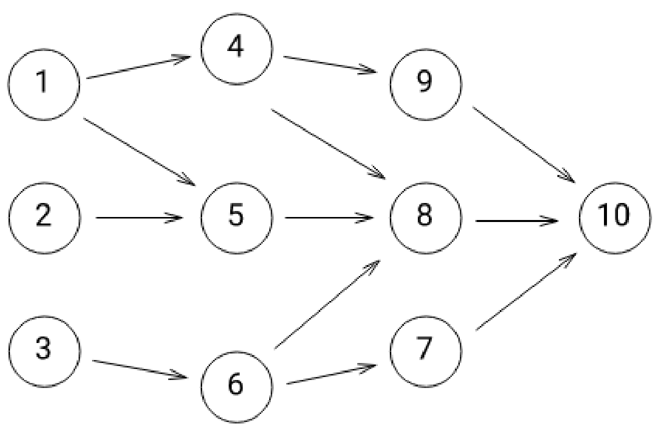


We can see from the diagram that the pipeline exhibits the requirements of a DAG. First, there are a set of vertices and edges, second, there is a direction of each task, and finally, there are no cycles. But just because the pipeline can be written as a DAG, why does that mean we should write it as such?

The reason is that the DAG structure is built in a way that naturally creates an efficient ordering of dependent tasks. There is a DAG sorting algorithm for exposing this order that we can take advantage of when scheduling our tasks. We'll see that using a DAG, we can implement task scheduling in linear time, O(V+E), where V and E are the number of vertices and edges).

A graph structure does not always start with a *single* root. Instead, there could be multiple starting points in a DAG:

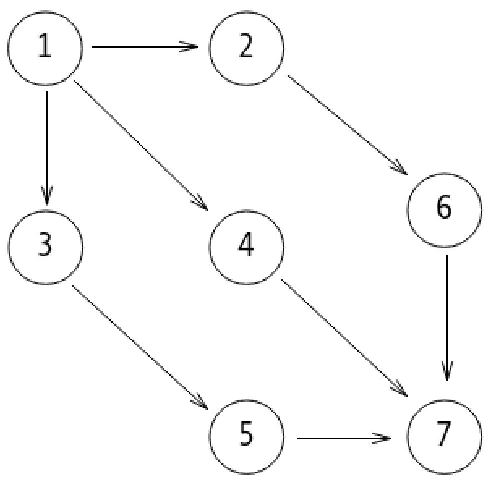
3 Root DAG



Let's see if there's another data structure we could use. First, we need the ability to *link* vertices to multiple nodes in the graph. Then, we need to easily loop through these nodes to create our graph.

A simple data structure which provides us with each of those behaviours is a `dict` with `list` values. Here's how it could represent our example graph:

Directed Graph



```
graph = {
   # Node: List of nodes to.
   7: [],
   6: [7],
   5: [7],
   4: [7],
   3: [5],
   2: [6],
   1: [2, 3, 4]
}

class DAG(DQ):
   def __init__(self):
       self.graph = {}

   def add(self, node, to=None):
       if not node in self.graph:
           self.graph[node] = []
       if to :
           if not to in self.graph:
               self.graph[to] = []
           self.graph[node].append(to)


dag = DAG()
dag.add(1)
dag.add(1, 2)
dag.add(1, 3)
dag.add(1, 4)
dag.add(3, 5)
dag.add(2, 6)
dag.add(4, 7)
dag.add(5, 7)
dag.add(6, 7)
```

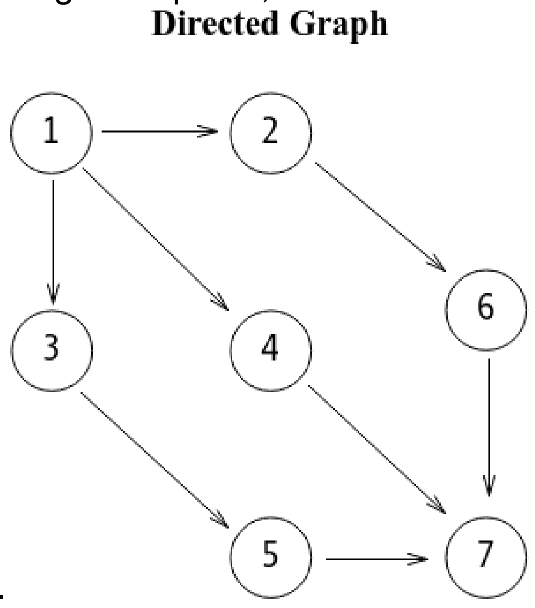
Sorting the DAG

What does it mean to sort the DAG? In our previous sorting algorithms, we compared object values, and sorted them in ascending or descending order. In a DAG, though, what does it mean to be in ascending or descending order?

Remember, our use case for the DAG was to place tasks in an order of *dependencies*. What we're looking for is an ordering of tasks that start with the most depended on tasks, and end with the least depended on. In our pipeline example, we're trying to order our tasks to start with parsing a file, and ending with summarising.

Let's take a look back at our DAG example. At a glance, it's reasonable to assume that the node that is depended on the most is the first node, 1.

Then, following the paths, we can see that each node decreases in importance.



Another way to phrase this, is that the longer the path to the node, the less that node is depended on. Take a look at 1, the most dependent, it has the shortest directed path: 0 steps. Conversely, the largest directed path is 7 with 4 steps, and it is the least depended on.

Using this hypothesis, we can perform the following: 1. Find the "root" nodes of the graph with 0 dependencies. 2. For each node, find the longest path from the node to the roots. 3. Sort by the longest paths.

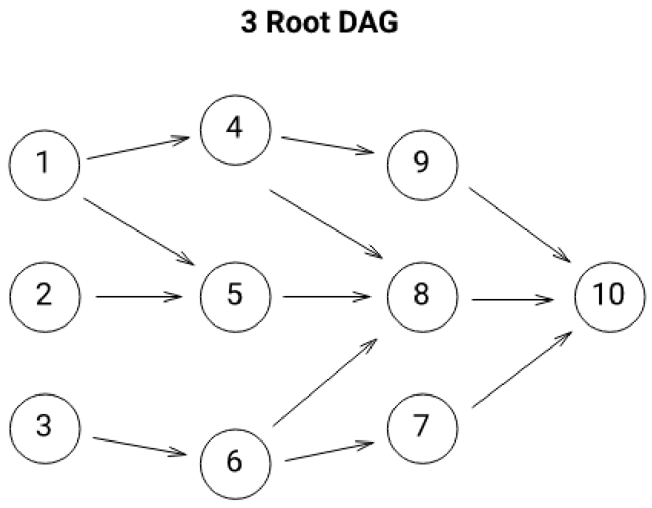
This seems reasonable, but there's some major time complexity drawbacks here. For points 2, and 3, the time complexities of each one are O(n2) (longest path) and O(log(n)) (fastest path) respectively! That's a worst case of O(n2)

Finding Number of In Degrees

The previous screen's longest path hypothesis wasn't wrong, it was the proposed algorithm that would have taken too long. In the next few screens, we'll develop an efficient algorithm to sort the DAG.

First, to find the longest path, it's necessary to know which nodes "start" the directed graph. By start, we mean the root nodes that the graph expands from. The question is, what determines a root node?

Let's look at the following graph:



Clearly, the root nodes of the graph are 1, 2, and 3. But what makes the root nodes different from every other node in the graph? The answer is, the number of in-degrees.
The number of in-degrees is the total count of edges pointing towards the node. For example, node 5 has 2 in-degrees, and node 8 has 3. For each root node, however, the number of in-degrees will always be 0.

```
class DAG(BaseDAG):
    def in_degrees(self):
        self.degrees = {}
        for node  in self.graph:
            if node not in self.degrees:
                self.degrees[node] = 0
            for pointed in self.graph[node]:
                if pointed not in self.degrees:
                    self.degrees[pointed] = 0
                self.degrees[pointed] += 1



dag = DAG()
dag.add(1)
dag.add(1, 2)
dag.add(1, 3)
dag.add(1, 4)
dag.add(3, 5)
dag.add(2, 6)
dag.add(4, 7)
dag.add(5, 7)
dag.add(6, 7)
dag.in_degrees()
```

Challenge: Sorting Dependencies

```
from collections import deque
```

```
class DAG(BaseDAG):
    def sort(self):
        self.in_degrees()
        root_node=deque()
        for node in self.graph:
            if self.degrees[node]==0:
                #print("{} is a root_node".format(node))
                root_node.append(node)

        searched =[]

        while root_node:
            rn = root_node.popleft()
            for pointer in self.graph[rn]:
                self.degrees[pointer] -=1
                if self.degrees[pointer] == 0:
                    root_node.append(pointer)

            searched.append(rn)

        return searched

dag = DAG()
dag.add(1)
dag.add(1, 2)
dag.add(1, 3)
dag.add(1, 4)
dag.add(3, 5)
dag.add(2, 6)
dag.add(4, 7)
dag.add(5, 7)
dag.add(6, 7)
dependencies = dag.sort()
```

Enhance the Add Method

The algorithm we wrote in the previous screen is called a topological sort. Specifically, the algorithm we implemented was called Kahn's Algorithm, a famous DAG sorting algorithm. An interesting property about this sorting algorithm is that it can also determine if a graph has a cycle or not.

To test for cyclicity we first sort the DAG, return its topologically sorted list of visited nodes, and then check the length of sorted nodes. If the length of the sorted nodes is greater than the length of the nodes in the graph, then there must be a cycle! Because the topological sort visits all pointed nodes, if there is a cycle, we will be visiting a previous node making the visited list greater than the number of vertices in the graph.

For robustness, we should not add a node to the DAG if it makes it cyclical.

```
class DAG(BaseDAG):
    def add(self, node, to=None):
        if not node in self.graph:
            self.graph[node] = []
        if to:
            if not to in self.graph:
                self.graph[to] = []
            self.graph[node].append(to)
        # Add validity check.
        if len(self.sort()) != len(self.graph):
            raise Exception

dag = DAG()
dag.add(1)
dag.add(1, 2)
dag.add(1, 3)
dag.add(1, 4)
dag.add(3, 5)
dag.add(2, 6)
dag.add(4, 7)
dag.add(5, 7)
dag.add(6, 7)
# Add a pointer from 7 to 4, causing a cycle.
dag.add(7, 4)
```

Challenge: Running the Pipeline

With the tasks added in order, it's time to run the pipeline. The major thing to notice is that there's no input for the run function. To run, we create a task that begins the pipeline by returning a static object (in our case, `first()` returns 20).

Furthermore, there is no concept of a "last task" in a DAG. Therefore, the way we can represent our tasks, and their outputs during a run, is by using a dictionary that maps `function: output`. With this dictionary, we can store outputs after task completion so we can use them as inputs for the next tasks that require them.

```
class Pipeline(DQ):
    def __init__(self):
        self.tasks = DAG()
    def task(self, depends_on=None):
        def inner(f):
            self.tasks.add(f)
            if depends_on:
                self.tasks.add(depends_on, f)
            return f
        return inner

    def run(self):
        visited = self.tasks.sort()
        completed = {}
        for task in visited:
            for node, values in self.tasks.graph.items():
                if task in values:
                    completed[task] = task(completed[node])
            if task not in completed:
                completed[task] = task()
        return completed


pipeline = Pipeline()

@pipeline.task()
def first():
    return 20

@pipeline.task(depends_on=first)
def second(x):
    return x * 2

@pipeline.task(depends_on=second)
def third(x):
    return x // 3

@pipeline.task(depends_on=second)
def fourth(x):
    return x // 4

outputs = pipeline.run()
```