Python 1.4

Summary: This document provides basic information about the extension providing **Python** support.

Extension ID

com.castsoftware.python

What's new?

Please see **Python 1.4 - Release Notes** for more information

Description

This extension provides support for **Python**.

In what situation should you install this extension?

If your application contains **Python** source code (both **.py** and **.jy** extensions are supported) and you want to view these object types and their links with other objects, then you should install this extension.

Files analyzed

Icons	File	Extension	Note
	Python	.py,	Python files - standard extension.
	Jython	.jy	By convention, Python files to be run in a Java implementation of the Python interpreter.
-	**YAML (YAML Ain't Markup Language)**	*.yml, *.yaml,	Files related to the **YAML** language, commonly used for configuration purposes. Necessary to interpret Amazon Web Services deployment code.

Supported Python versions

The following table displays the supported versions matrix:

Version	Support
3.x	✅
2.x	✅
1.x	❌

Function Point, Quality and Sizing support

This extension provides the following support:

- **Function Points (transactions)**: a green tick indicates that OMG Function Point counting and Transaction Risk Index are supported
- **Quality and Sizing**: a green tick indicates that CAST can measure size and that a minimum set of Quality Rules exist

Function Points (transactions)	Quality and Sizing	Security
✅	✅	✅

AIP Core compatibility

This extension is compatible with:

CAST AIP release	Supported
8.3.x	✓

Supported DBMS servers

This extension is compatible with the following DBMS servers:

DBMS	Supported
CSS / PostgreSQL	✓

Prerequisites

✓	An installation of any compatible release of AIP Core (see table above)

Dependencies with other extensions

Some CAST extensions require the presence of other CAST extensions in order to function correctly. The **Python** extension requires that the following other CAST extensions are also installed:

- **Web Services Linker** (internal technical extension)
- **CAST AIP Internal extension** (internal technical extension)

> Note that when using the **CAST Extension Downloader** to download the extension and the **Manage Extensions** interface in **CAST Server Manager** to install the extension, any dependent extensions are **automatically** downloaded and installed for you. You do not need to do anything.

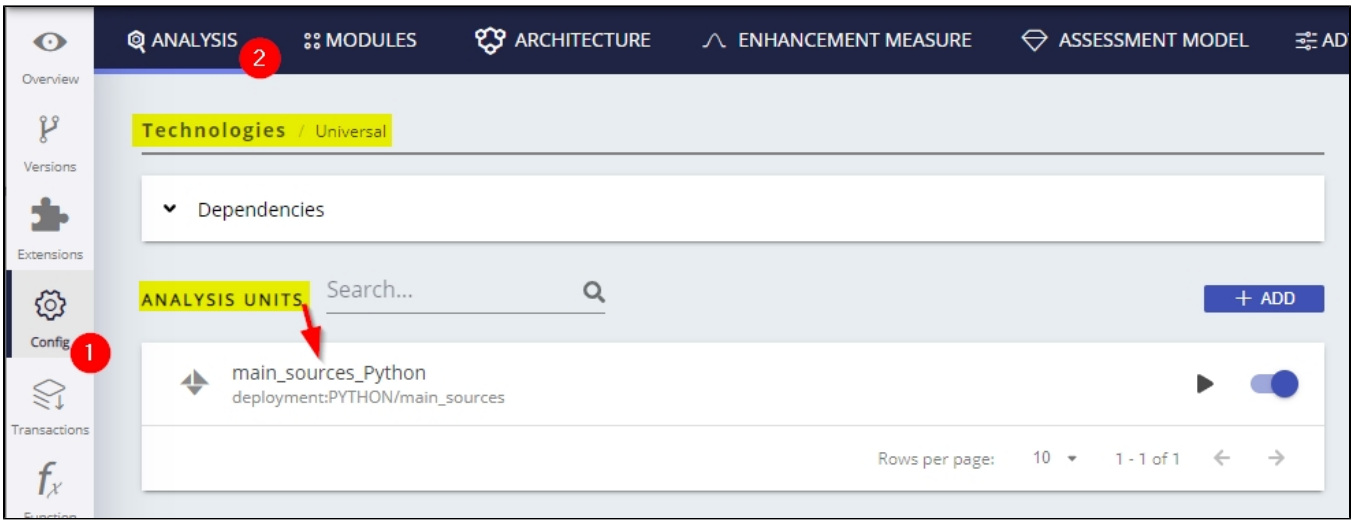
Download and installation instructions

The extension will be automatically downloaded and installed in AIP Console when you deliver Python code. You can also **manually install** the extension using the **Application - Extensions** interface. When installed, follow the instructions below to run a new analysis/snapshot to generate new results:

- **Advanced onboarding - run and validate the initial analysis**
- **Advanced onboarding - snapshot generation and validation**

Source code discovery

A discoverer is provided with the extension to automatically detect Python code: a Python project will be discovered for the package's root folder when at least one **.py or .jy (jython) file** is detected in the root folder or any sub-folders. For every Python project located, **one Universal Technology Analysis Unit** will be created:



Analysis - Automatic skipping of unit-test code and external libraries

The analyzer skips files that are recognized as forming part of testing code, i.e., in principle, code not pertaining to production code. The reason to avoid inclusion of testing code is that many Quality Rule violations are overrepresented in test code, either because code tends to be of poorer quality (certainly not critical) or prevalence of particular testing patterns. Accounting for test code would negatively impact the total score of the project.

Similarly we skip folders that contain external python libraries. Currently we only skip the canonical folders *site-packages* and *dist-packages* (the latter being used in certain Linux distributions). Not only analyzing external libraries is discouraged, but it can interfere with correct interpretation of supported libraries and frameworks, and have a serious impact in memory consumption and overall analysis performance.

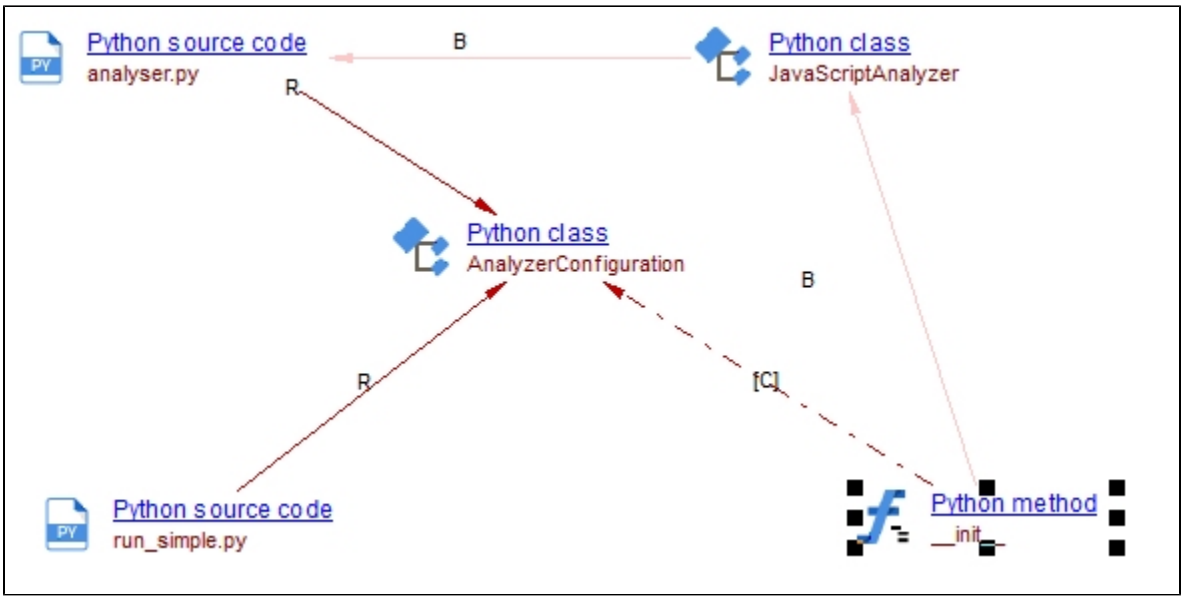
The heuristics used by the analyzer are based on detecting unit-test library imports, and file and path naming conventions as summarized in the table below:

Type	Value	HeaderLines	MinimumCount
Type	Value	HeaderLines	MinimumCount
FilePath	**/test_*.py		
FilePath	**/*_test.py		
FilePath	**/*_test_*.py		
FilePath	**/test/*.py		
FilePath	**/tests/*.py		
FileContent	import unittest	12	
FileContent	from unittest import	12	
FileContent	from nose.tools import	12	
FileContent	self.assert		2
FilePath	**/site-packages/**		
FilePath	**/dist-packages/**		

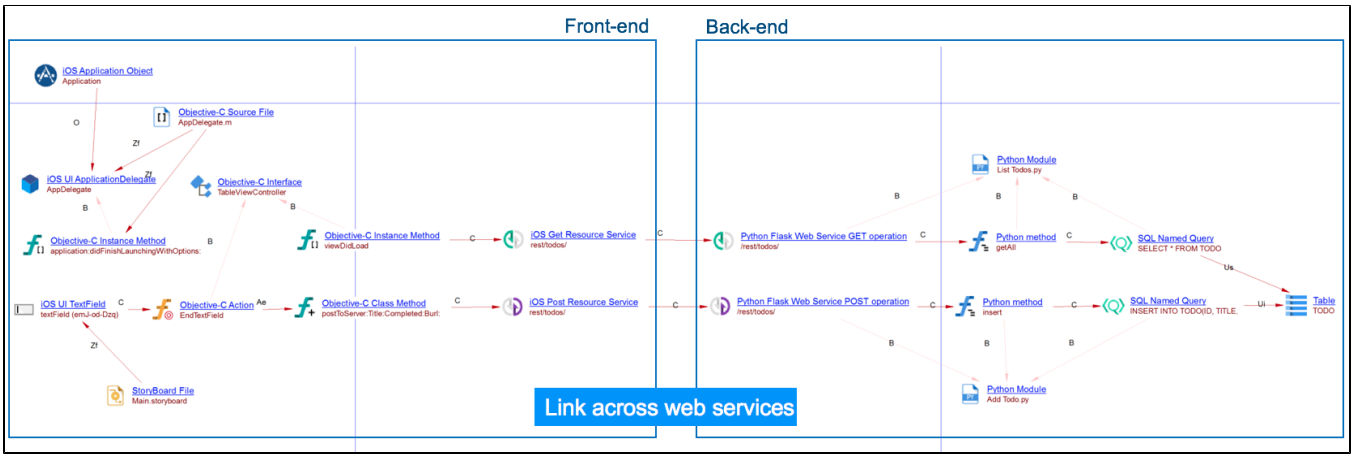
- The ** symbol represents any arbitrary path string, whereas * represents any string without directory slashes.
- The heuristics above should also similarly valid for .jy (jython) files.

What results can you expect?

Once the analysis/snapshot generation has completed, you can view the results in the normal manner:



Python Class and method example



iOS Front-end connected to a Python Flask Back-end.

Objects

The following specific objects are displayed in CAST Enlighten:

Icon	Description
	Python Project, Python External Library
	Python Module
	Python Class
	Python Method
	Python Script

	Python Get Urllib, Urllib2, Httplib, Httplib2, aiohttp Service Python *Flask, aiohttp* Web Service Get Operation
	Python Post Urllib, Urllib2, Httplib, Httplib2, aiohttp Service Python *Flask, aiohttp* Web Service Post Operation
	Python Put Urllib, Httplib, Httplib2, aiohttp Service Python *Flask, aiohttp* Web Service Put Operation
	Python Delete Urllib, Httplib, Httplib2, aiohttp Service Python *Flask, aiohttp* Web Service Delete Operation
	Python Query, Python ORM Mapping, Python File Query
	RabbitMQ Python QueueCall ActiveMQ Python QueueCall IBM MQ Python QueueCall
	RabbitMQ Python QueueReceive ActiveMQ Python QueueReceive IBM MQ Python QueueReceive
	Python Call To Java Program
	Python Call To Generic Program
Amazon Web Services	
	Python Call to AWS Lambda Function
	Python AWS Lambda GET Operation
	Python AWS Lambda POST Operation
	Python AWS Lambda PUT Operation

	Python AWS Lambda DELETE Operation
	Python AWS Lambda ANY Operation
	Python AWS SQS Publisher
	Python AWS SQS Receiver
	Python AWS SQS Unknown Publisher
	Python AWS SQS Unknown Receiver

Python callable artifact

Python Script, Python Module and *Python Method* objects form part of Python (callable) artifacts.

Links

The following links are created:

- **call links** between methods
- **inherit link** between hierarchically related classes
- **refer link** from methods to class (constructor call)
- **use link** between modules through import
- **call links** between Python callable artifacts and Python Call objects
- **call links** between Python Call objects and external programs or lambda functions

The following links are created between Python ORM Mapping objects and database table objects:

- **useSelectLink** in case of SELECT operation
- **useDeleteLink** in case of DELETE operation
- **useInsertLink** in case of INSERT operation
- **useUpdateLink** in case of UPDATE operation
- **call links** in case of generic operation on S3 buckets

Structural Rules

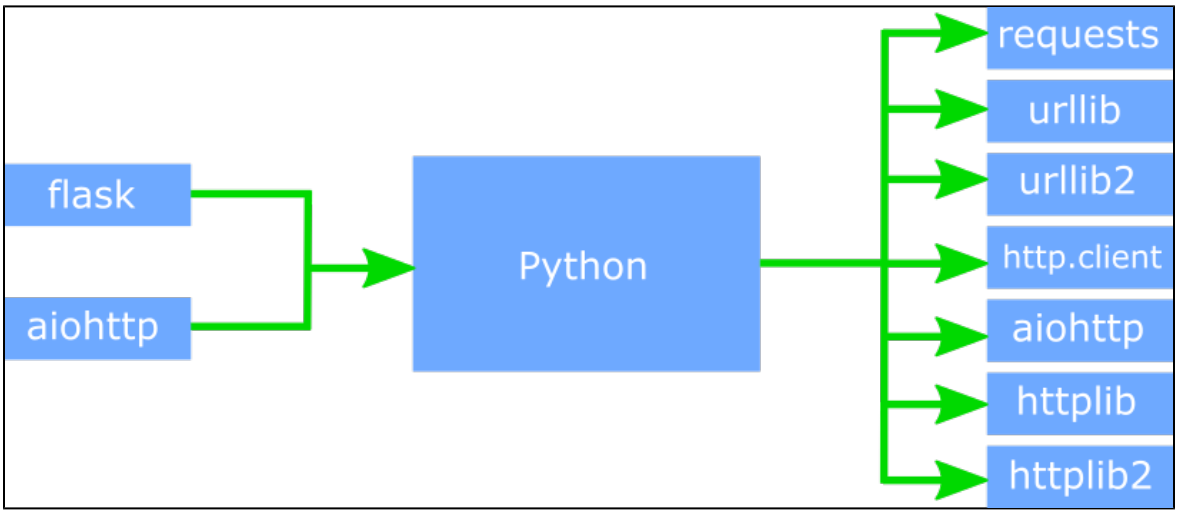
The following structural rules are provided:

1.4.0-beta3	**https://technologies.castsoftware.com/rules?sec=srs_python&ref=||1.4.0-beta3**
1.4.0-beta2	**https://technologies.castsoftware.com/rules?sec=srs_python&ref=||1.4.0-beta2**
1.4.0-beta1	**https://technologies.castsoftware.com/rules?sec=srs_python&ref=||1.4.0-beta1**
1.4.0-alpha2	**https://technologies.castsoftware.com/rules?sec=srs_python&ref=||1.4.0-alpha2**
1.4.0-alpha1	**https://technologies.castsoftware.com/rules?sec=srs_python&ref=||1.4.0-alpha1**

You can also find a global list here: **https://technologies.castsoftware.com/rules?sec=t_1021000&ref=||**

Web Service calls and operations support

The following libraries are supported for Web Service operations (left) and Web Service HTTP API calls (right):



Once the Python extension analysis is finished, the analyzer will output the final number of web service call and operation objects created.

requests

Example for GET request:

```
import requests
r = requests.get('https://api.github.com/events')
```

urllib

Example for GET request:

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
   html = response.read()
```

urllib2

Example for GET request:

```
import urllib2

req = urllib2.Request('http://python.org/')
response = urllib2.urlopen(req)
the_page = response.read()
```

Example for POST request.

```
import urllib2
import urllib

values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.urlencode(values)

req = urllib2.Request('http://python.org/', data)
response = urllib2.urlopen(req)
the_page = response.read()
```

PUT and DELETE calls are not supported by the urllib2 module (**Python version 2.x**) by default. Workarounds to bypass this limitation are not detected by the analyzer.

urllib3

Example for GET request:

```
# using PoolManager
import urllib3
http = urllib3.PoolManager()
r = http.request('GET', 'http://httpbin.org/robots.txt')

# using HTTPConnectionPool
import urllib3
pool = urllib3.HTTPConnectionPool()
r = pool.request('GET', 'http://httpbin.org/robots.txt')
```

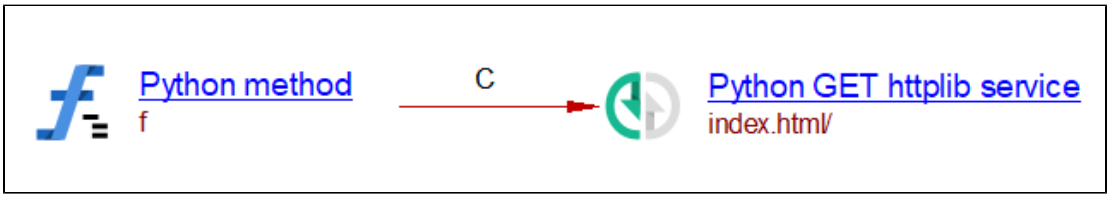
Note: The urllib3 web service object is represented with the same *Python GET urllib service* as that used for urllib.

httplib

Example for GET request:

```
from httplib import HTTPConnection
def f():
    conn = HTTPConnection("www.python.org")
    conn.request("GET", "/index.html")
```

Example link from method "f" to the get httplib service:



http.client

Example for GET request:

```
from http.client import HTTPConnection
def f():
    conn = HTTPConnection("www.python.org")
    conn.request("GET", "/index.html")
```

In this case a *Python Get Httplib Service* will be generated (the *httplib* module from Python 2 has been renamed to *http.client* in Python 3).

httplib2

The following code will issue a http get to the url 'https://api.github.com/events':

```
import httplib2
h = httplib2.Http(".cache")
(resp, content) = h.request("https://api.github.com/events")
```

aiohttp

The following code will issue a http get to the url 'https://api.github.com/events':

```
import aiohttp
session = aiohttp.ClientSession()
res = session.get('https://api.github.com/events'
```

The *aiohttp* module can be also used in server mode, implementing web service operations

```
from aiohttp import web
async def handler(request):
    return web.Response(text="Welcome in Python")
app = web.Application()
app.router.add_get('/index', handler)
web.run_app(app)
```

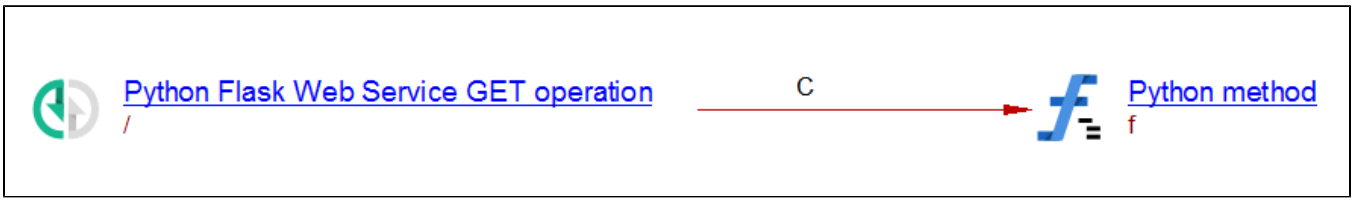
In this case a Web Service Operation object associated to the function (coroutine) *handler* will be generated similar to the example for flask given below.

flask

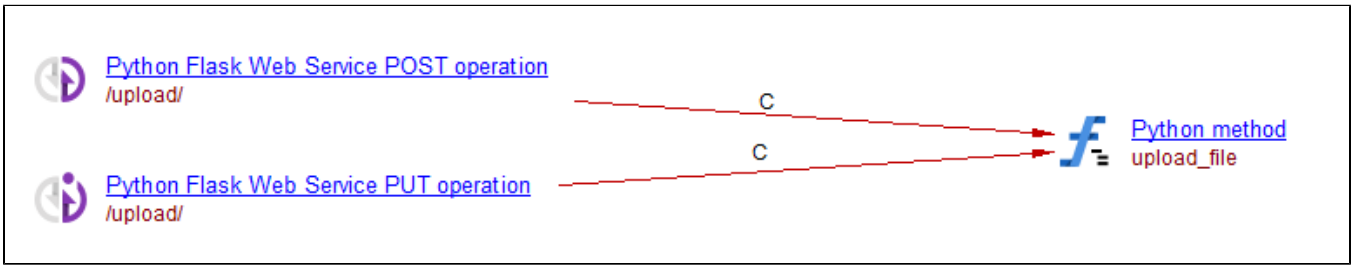
Flask *route* annotations for web service operations (GET, PUT, POST, DELETE) are supported. In particular, any decorator with the format *@prefix*.route is considered as a flask annotation where *prefix* can be a Flask application object or blueprint object. In the following example, a default GET operation is ascribed to the function *f,* and the POST and PUT operations to the *upload_file* function:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def f():
    return 'hello world!'

@app.route('/upload', methods=['POST', 'PUT'])
def upload_file()
        if request.method == 'POST':
            pass
        # ...
```

The link between the GET operation named after the routing URL "/" and the called function *f* is represented by an arrow pointing to the function:

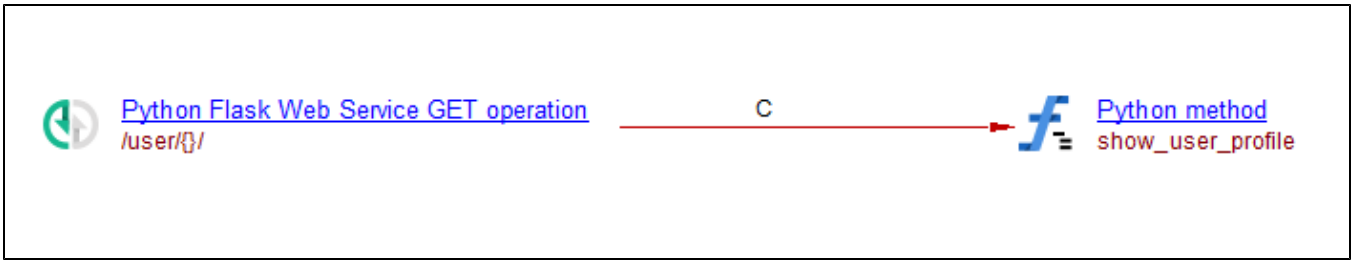


The name of a saved Web Service Operation object will be generated from the routing URL by adding a final slash when not present. In this example the name of the PUT and POST operations is "/upload**/**" after the routing url "/upload".



URL query parameters such as @app.route('/user/**<username>**') are supported. In this case the generated Web Service Operation object will be named as **/user/{}/**, as shown in the example below.

```
from flask import Flask
app = Flask(__name__)

@app.route('/user/<username>')
def show_user_profile(username):
    return 'User %s' % username
```



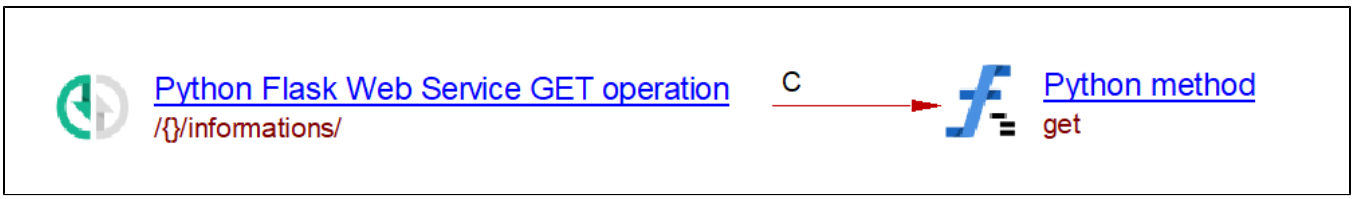
Similarly double slashes **//** in flask routing URLs are transformed into **/{}/**. Additional backslashes inside URL query parameters of type *path* [@app.route('/<path:path>')] are not resolved (which in principle could catch any URL) so the web service will be named as a regular parameter **/{}/.**

The equivalent alternative to routing annotations using the Flask *add_url_rule* is also supported.

```
from flask import Flask
app = Flask(__name__)

def index():
    pass

app.add_url_rule('/', 'index')
```

Plugable views are also supported for Flask *add_url_rule.*

```
from flask.views import MethodView

class InformationAPI(MethodView):

    def get(self):
        information = Information.from_data(request.data)
        ...

app.add_url_rule('/<info>/informations/', view_func=InformationAPI.as_view('informations'))
```



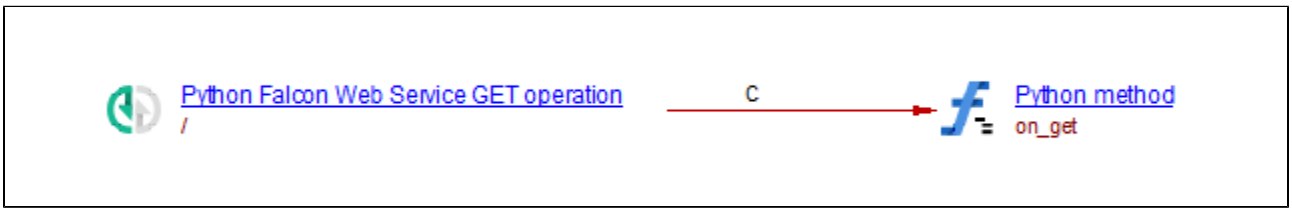
falcon

Falcon *route*annotations for web service operations (GET, PUT, POST, DELETE) are supported.

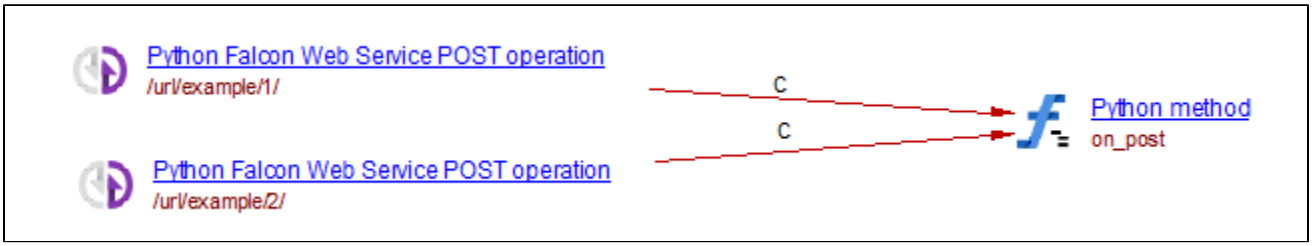
In the following example, a default GET operation is ascribed to the function*on_get* from *GetResource*class,and the POST and PUT operations to the *on_put*and*on_post*functions from*Put_PostResource*with two differents urls routing:

```
importfalcon

classGetResource():
defon_get():
print('on_get function')

classPut_PostResource():
defon_put():
print('on_get function')
defon_post():
print('on_get function')

app=falcon.App()

app.add_route('', GetResource())
app.add_route('/url/example/1', Put_PostResource())
app.add_route('/url/example/2', Put_PostResource())
```

The link between the GET operation named after the routing URL "/" and the called function*on_get* is represented by an arrow pointing to the function:



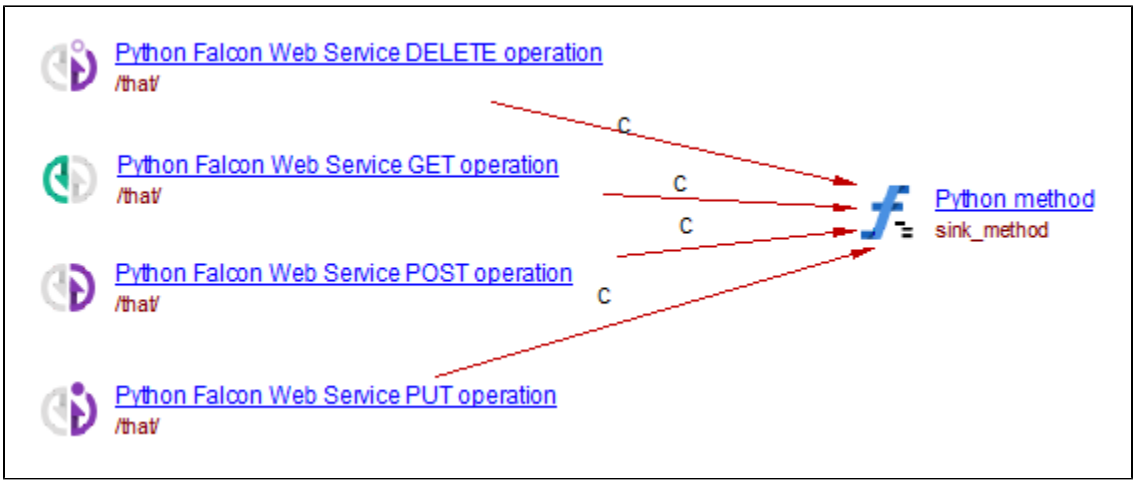
The name of a saved Web Service Operation object will be generated from the routing URL by adding a final slash when not present. In this example the name of the POST operations is "/url/example/1**/**" and "/url/example/2**/**" after the routing url "/url/example/1" and "/url/example/2".



Sinks are supported with the following rules : If no route matches a request, but the path in the requested URI matches a sink prefix, Falcon will pass control to the associated sink, regardless of the HTTP method requested. If the prefix overlaps a registered route template, the route will take precedence and mask the sink.

In this case Web Service Operation objects generated as sinks will be named as**/that/**, and not as**/this/**since another Web Service Operation object exists with an overlapping url.

```
importfalcon

app=falcon.App()

classGetResource():
defon_get():
print('on_get function')

defsink_method(resp,**kwargs):
resp.body="Sink"
pass

app.add_route('this/is/the/way', GetResource())
app.add_sink(sink_method, prefix='/that')#get, post, put & delete routes will be created and linked to
sink_method
app.add_sink(sink_method, prefix='/this')#no routes created because Url overlaps another route
```



The optionnal *suffix*keyword argument of Falcon *add_route* is supported. In this way, multiple closely-related routes can be mapped to the same resource.

```
importfalcon
app=falcon.App()

classPrefixResource(object):

defon_get(self, req, resp):
pass

defon_get_foo(self, req, resp):
pass

defon_post_foo(self, req, resp):
pass

defon_delete_bar(self, req, resp):
pass

app.add_route('get/without/prefix', PrefixResource())
app.add_route('get/and/post/prefix/foo', PrefixResource(), suffix='foo')
app.add_route('delete/prefix/bar', PrefixResource(), suffix='bar')
```

web2py

Example for GET request:

```
from gluon.tools import fetch
page = fetch('http://www.google.com/')
```

Database access

PEP 249

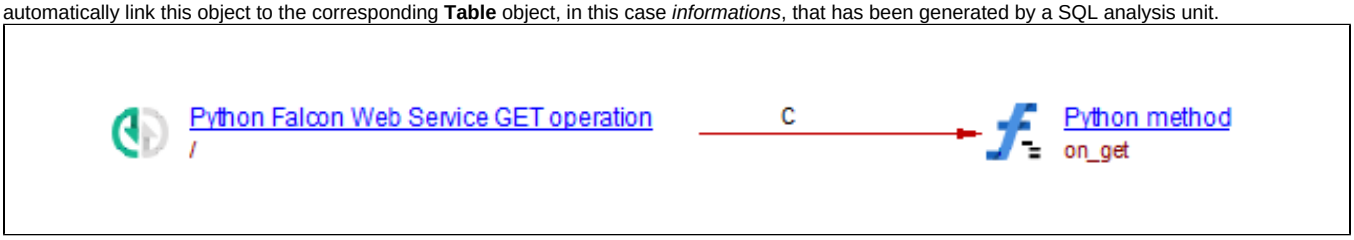
Simple database queries consistent with the Python Database API Specification (PEP 249) are recognized. This allows to support a large number of important libraries interfacing Python and SQL databases (SQLite, MySQL, etc). The analyzer identifies *execute* method calls as potential database queries and searches for generic SQL statements passed in as an argument ('SELECT ...", "INSERT ...)". In the example below data from the *stocks* table is retrieved via a SELECT statement passed explicitly by a string to the execute method of a cursor object.

```
# query.py
import sqlite3

conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute('SELECT * FROM informations')
```

In addition to *execute* method calls, the analyzer identifies *raw* method calls which are used in Django framework. SQL queries can be defined directly or via a method.

```
from django.db import models
...
def function(self):
    sql = 'SELECT * FROM informations'
    return model.objects.raw(sql)
```

The analyzer creates a **Python Query** object with name *SELECT * FROM informations* (first 4 words are only used as naming convention) representing a call to a database. Provided analysis **dependencies** between SQL and Python are configured in CAST Management Studio, the analyzer will automatically link this object to the corresponding **Table** object, in this case *informations*, that has been generated by a SQL analysis unit.



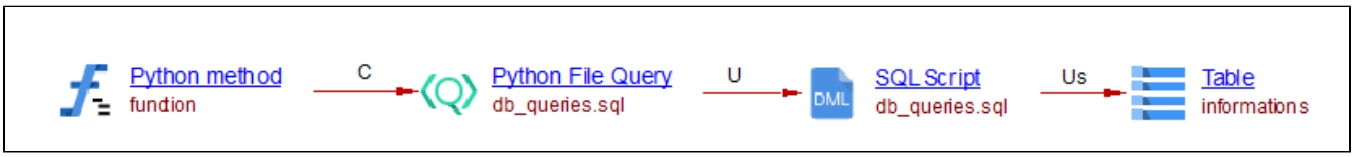
In some cases SQL queries can be defined via SQL files.

```
def function(self):
        file_path = "db_queries.sql"
        sql = open(file_path).read()
        cursor.execute(sql)
```

where the file *db_queries.sql* contains SQL code that is analyzed independently by the sqlanalyzer extension.

```
CREATE TABLE IF NOT EXISTS informations;
SELECT * FROM informations
```

In this situation, the analyzer will create a **Python File Query** object with the name of the sql file. This object will make the link between the method containing the query and the SQL script (if it is present, and **dependencies** between SQL and Python are configured as previously mentioned), so that the end point of the transaction (for example, a table) can be reached.



Only files containg '.sql' extensions are supported.

SQLAlchemy

SQLAlchemy is a Python SQL toolkit providing a way of interaction with databases. SQLAlchemy includes both a database server independent SQL expression language and an Object Relational Mapper (ORM). An ORM presents a method of associating user-defined Python classes with database tables and instances of the classes(objects) with rows in their corresponding tables. The analyzer identifies *query* method calls in addition to *execute* method calls.

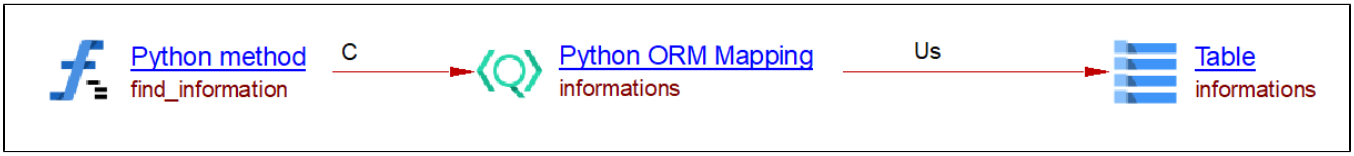
Example using *query* method call:

```
class UserTable:
    __tablename__ = "users"

    def __init__(self):
        pass

class User(UserTable):
    __tablename__ = "users_table"

    def __init__(self):
        UserTable.__init__()

    def f(self):
        query = UserTable.query().filter(UserTable.name == "new_user") #query().filter(...) is equivalent to
SELECT statement
```

Example using *execute* method call:

```
class Information:
    __tablename__ = "informations"

    def find_information(self):
        informations_table = Information.__table__
        select_query = (
            informations_table.filter(informations_table.id == target.information_id)
            )
        connection.execute(select_query)
```

In this example the analyzer creates a **Python ORM Mapping** object with the name of the table designated by *__tablename__* in class. As in the case of creation of Python Query objects, it is assumed that analysis dependencies between SQL and Python are correctly configured in CAST Management Studio. Then, links between these objects and the corresponding **Table** objects (in this example *informations,* generated by a SQL analysis unit) will automatically be created by the analyzer. The type of the link in this particular case is *useSelectLink* (**Us**) because of the *filter()* method call present in the query expression.

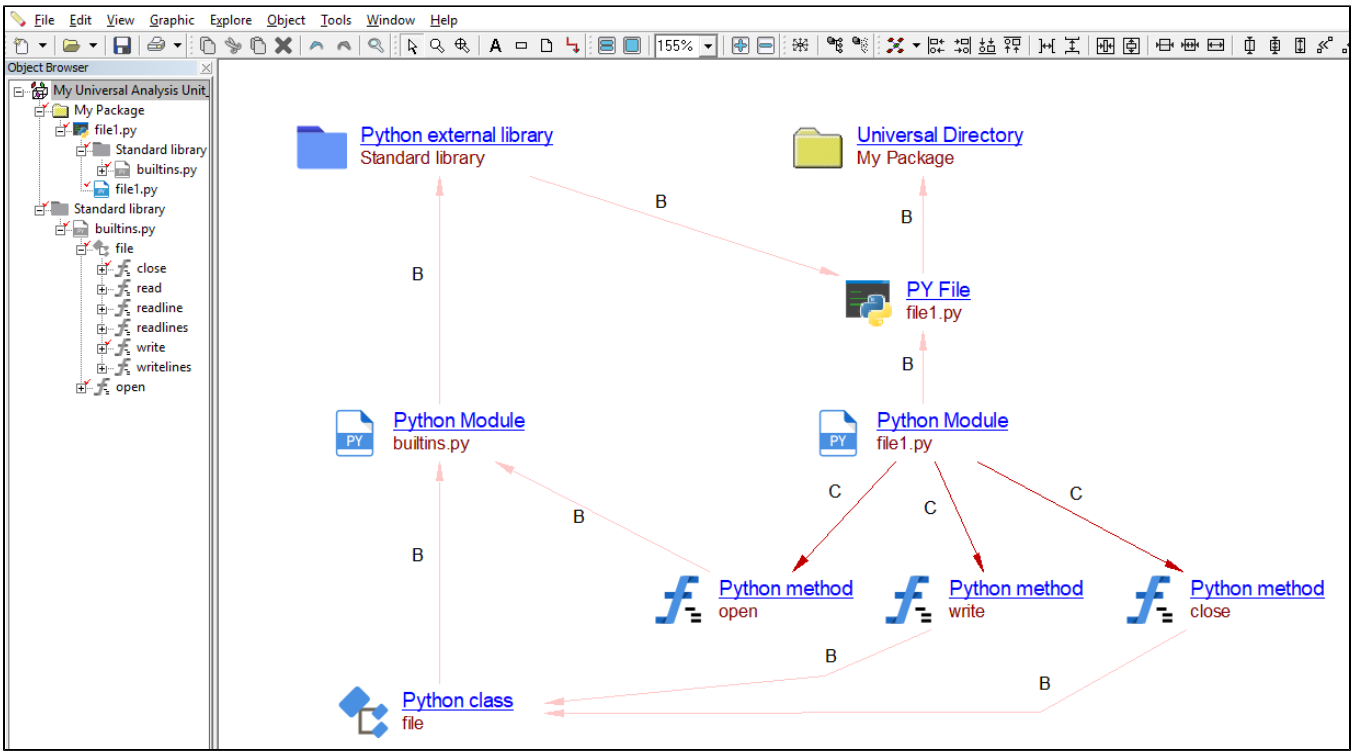


File system access functions

Representing end points based on file system access is important to automatically configure transactions. Towards this goal we aim at providing automatic recognition of standard library input/output functions. Currently we provide support for the built-in **open** function and the most common methods associated to **file**-like objects **write**, **read**, **writelines**, **readline**, **readlines**, and **close**, as shown in the example below.

```
# file1.py

data = """<html>
<header><title>This is title</title></header>
<body>
Hello world
</body>
</html>
"""

f = open('page.html', 'w')
f.write(data)
f.close()
```

The objects corresponding to the code of the *file1.py* file are inside the *Universal Directory* root object. Additionally the analyzer will automatically generate a *Python external library* object representing the Python *Standard Library*. Within this, the Python built-in library is abstracted as a *Python source code* object named *builtins.py,* with its corresponding *open* function and *file* class (abstracting general file-like objects) that contains the above mentioned methods. No differences are considered between Python2 and Python3 built-in functions. Notice the **external** character of these objects denoted by gray-shaded icons in the left Object Browser panel.



Due to implementation constraints in CAIP versions [7.3.6, 8.1] a spurious link is generated between the *Python external library* object and a *PY File* object.

Message Queues support

Introduction

Message queues are software-engineering components used for inter-process communication, or for inter-thread communication within the same process. They use a queue for messaging. A producer posts messages to a queue. At the appointed time, the receivers are started up and process the messages in the queue. A queued message can be stored and forwarded, and the message can be redelivered until the message is processed. Message queues enable asynchronous processing, which allows messages to be queued without the need to process them immediately.

Message Queues currently handled by the Python analyzer

ActiveMQ

Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (**JMS**) client. The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible. ActiveMQ acts as the middleman allowing heterogeneous integration and interaction in an asynchronous manner.

IBM MQ

IBM MQ is a family of network message-oriented middle ware products that IBM launched. It was originally called **MQSeries** (for "Message Queue"), and was renamed *WebSphere MQ* to join the suite of WebSphere products. IBM MQ allows independent and potentially non-concurrent applications on a distrib uted system to securely communicate with each other. IBM MQ is available on a large number of platforms (both IBM and non-IBM), including z/OS (mainfr ame), OS/400 (IBM System i or AS/400), Transaction Processing Facility, UNIX, Linux, and Microsoft Windows.

RabbitMQ

RabbitMQ is an open source message-queueing software called a message broker or queue manager RabbitMQ implements **AMQP**. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.

Message queue applications using the below mentioned frameworks/clients are handled:

- **Library** interface with **STOMP** protocol for ActiveMQ
- **Pika** client with **AMQP** protocol for RabbitMQ
- **MQ-Light** client with **TCP/IP** for IBM MQ
- **Pymqi** python extension for IBM MQ

CAST Enlighten screenshots

When a message queue application is analyzed by the Python analyzer, the following transactions can be found at the end of analysis:

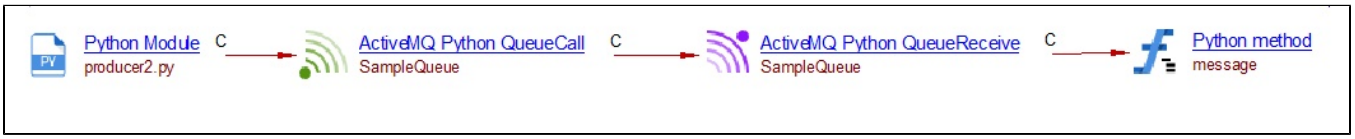
Example of ActiveMQ Producer

```
import stomp

conn = stomp.Connection10()
conn.start()
conn.connect()
conn.send('SampleQueue', 'Its working!!')
conn.disconnect()
```

Example of ActiveMQ Consumer

```
import stomp

queue = 'SampleQueue'
conn = stomp.Connection10()
conn.start()
conn.connect()
conn.subscribe(queue)
conn.disconnect()
```

CAST Enlighten screenshot of ActiveMQ Transaction



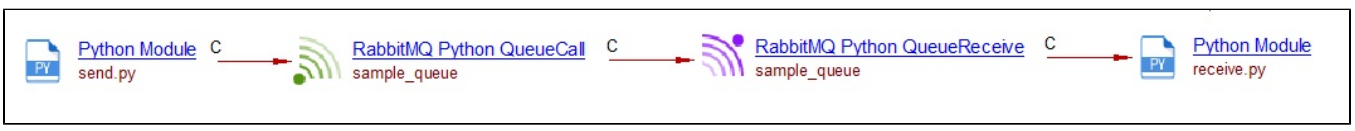
Example of RabbitMQ Producer

```
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.queue_declare(queue = "sample_queue")
channel.basic_publish(exchange = '', routing_key = "sample_queue", body = "Hello world!" )
connection.close()
```

Example of RabbitMQ Consumer

```
import pika

def callback(ch, method, properties, body):
    print("[x] Received % r" % body)

connectionconnection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue = "sample_queue")
channel.basic_consume(callback, queue = "sample_queue", no_ack = True)
channel.start_consuming()
```

CAST Enlighten screenshot of RabbitMQ Transaction



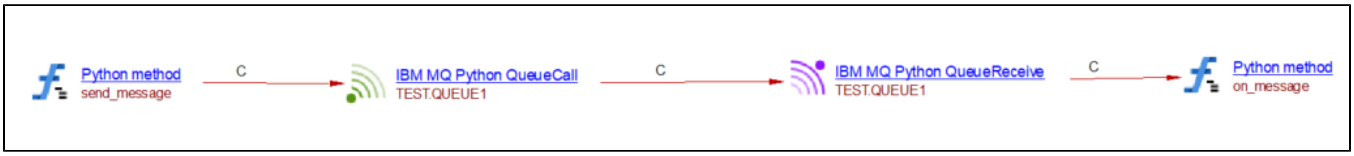
Example of IBM MQ Producer

```
import pymqi

def send_message(self):
    queue_manager = "QM01"
    channel = "SVRCONN.1"
    host = "192.168.1.135"
    port = "1434"
    queue_name = "TEST.QUEUE1"
    message = "Hello from Python!"

    qmgr = pymqi.connect(queue_manager, channel, conn_info)
    queue = pymqi.Queue(qmgr, queue_name)
    queue.put(message)
    queue.close()
    qmgr.disconnect()
```

Example of IBM MQ Consumer

```
import pymqi

def on_message(self,headers, msg):
    queue_manager = "QM01"
    channel = "SVRCONN.1"
    host = "192.168.1.135"
    port = "1434"
    queue_name = "TEST.QUEUE1"

    qmgr = pymqi.connect(queue_manager, channel, conn_info)
    queue = pymqi.Queue(qmgr, queue_name)
    message = queue.get()
    queue.close()
    qmgr.disconnect()
```

CAST Enlighten screenshot of IBM MQ Transaction



Amazon Web Services

The library **boto3** is supported, the AWS SDK for python (with certain limitations). Configuration YAML files are also analyzed in search of serverless deployment frameworks.

AWS Lambda in AWS deployment frameworks

Serverless framework, Serverless Application Model (SAM), and Cloudformation are supported. These are frameworks using *.yml and *.yaml (or *.json, currently not supported in this extension) file to set up AWS environment.
Whenever the runtime set in these files is **pythonX.Y**, the com.castsoftware.python extension is responsible for creating the corresponding Python AWS Lambda Function, Python AWS Lambda Operation (which represent AWS APIGateway events), and Python AWS Simple Queue objects.

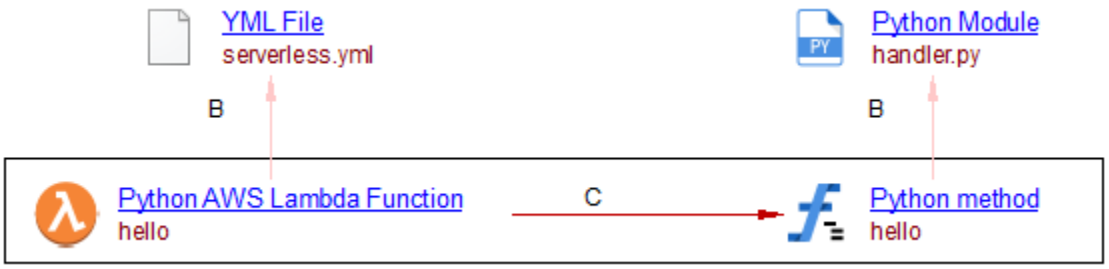
For example in the .yml deployment file below (taken from the Serverless examples for AWS) a Lambda function is defined (hello) and the handler's method name is referred:

```
service: aws-python # NOTE: update this with your service name

frameworkVersion: '2'


provider:
  name: aws
  runtime: python3.8
  lambdaHashingVersion: 20201221

functions:
  hello:
    handler: handler.hello
```

where the Python code of the handler:

```
# handler.py

def hello(event, context):
    body = {
        "message": "Go Serverless v2.0! Your function executed successfully!",
        "input": event,
    }

    return {"statusCode": 200, "body": json.dumps(body)}
```

The results in Enlighten:



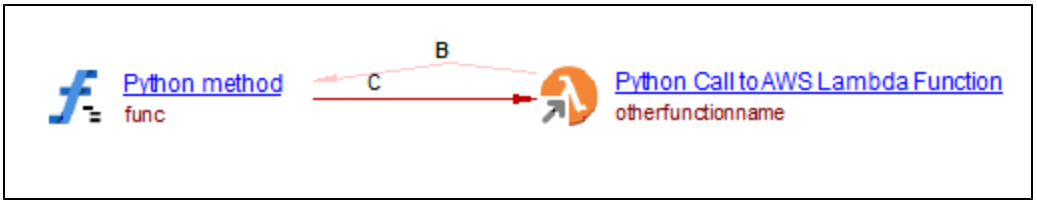
Boto3 AWS sdk for Python

Supported API methods (boto3)	Link Type	Caller	Callee
• botocore.client.Lambda.**invoke**	callLink	Python callable artifact	Python Call to AWS Lambda Function
• botocore.client.Lambda.**invoke_async**			

Example

A simple example showing representation of an invocation of a AWS Lambda function:

```
def func():
    lambda_client.invoke(FunctionName='otherfunctionname',
                         InvocationType='RequestResponse',
                         Payload=lambda_payload)
```



AWS SQS

Supported API methods (boto3)	Link Type	Caller	Callee
• botocore.client.SQS.**send_message** • botocore.client.SQS.**send_message_batch**	callLink	Python callable artifact	Python AWS SQS Publisher Python AWS SQS Unknown Publisher
• botocore.client.SQS.**receive_message**	callLink	Python AWS SQS Unknown Receiver Python AWS SQS Receiver	Python callable artifact

Code samples

In this code, the module *sqs_send_message.py* publishes a message into the "SQS_QUEUE_URL" queue and in *sqs_receive_message.py* is received:

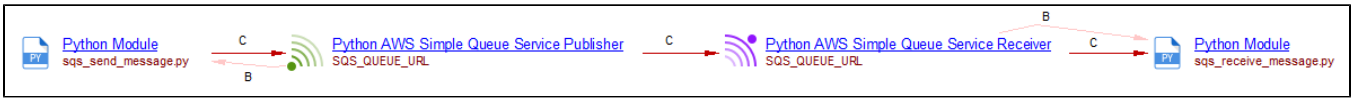
```
# Adapted from https://boto3.amazonaws.com/v1/documentation/api/latest/guide/sqs-example-sending-receiving-msgs.
html#example
# sqs_receive_message.py

import boto3

# Create SQS client
sqs = boto3.client('sqs')

queue_url = 'SQS_QUEUE_URL'

# Receive message from SQS queue
response = sqs.receive_message(QueueUrl=queue_url, ...)
```

and

```
# Adapted from https://boto3.amazonaws.com/v1/documentation/api/latest/guide/sqs-example-sending-receiving-msgs.
html#example
# sqs_send_message.py

import boto3

# Create SQS client
sqs = boto3.client('sqs')

queue_url = 'SQS_QUEUE_URL'

# Send message to SQS queue
response = sqs.send_message(QueueUrl=queue_url, ...)
```

The results derived from the analysis of the above code can be seen Enlighten:

Click to enlarge



Note: when the name of the queue passed to the API method calls is resolvable (either because of unavailability or because of technical limitations), the analyzer will create ***Unknown*** Publisher and Receive objects.

AWS S3

Supported API methods	Link Type	Caller	Callee	Other effects
botocore.client.S3.**create_bucket**	*N/A*	*N/A*	*N/A*	Creation of S3 bucket and S3 region objects
botocore.client.S3.**put_object**	useInsertLink	Python callable artifact	Python S3 Bucket, Python Unknown S3 Bucket	
botocore.client.S3.**delete_bucket**	useDeleteLink	Python callable artifact	Python S3 Bucket. Python Unknown S3 Bucket	
botocore.client.S3.**delete_object**				
botocore.client.S3.**delete_objects**				
botocore.client.S3.**get_object**	useSelectLink	Python callable artifact	Python S3 Bucket, Python Unknown S3 Bucket	
botocore.client.S3.**get_object_torrent**				
botocore.client.S3.**list_objects**				
botocore.client.S3.**list_objects_v2**				
botocore.client.S3.**put_bucket_logging**	useUpdateLink	Python callable artifact	Python S3 Bucket, Python Unknown S3 Bucket	
botocore.client.S3.**put_bucket_analytics_configuration**				

Note: CAST is considering the creation of generic 'callLinks' for the rest of the API methods acting on S3 buckets (https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#client).

Calls to external program from Python

Introduction

Python, often used to glue together different components of an application, provides various mechanisms to call external programs. By supporting these calls the analyzer can provide the linkage between different technology layers.

Supported API methods	Link Type	Caller	Callee
os.**system**	*callLink*	Python callable artifact	Python Call to Java Program, Python Call to Generic Program
os.**popen**			
subprocess.**call**			
subprocess.**check_call**			
subprocess.**run**			
subprocess.**Popen**			

Technologies currently handled by the Python analyzer

The Python analyzer currently supports calls to the following technologies

- Cobol
- Java: classes and .jar
- Python
- Shell

The Java technology is specific and has its own object because links are made using the fullname of the class, package and class name.

Furthermore, the link is not made to the class object but directly to its main method. Indeed, Java program can only be called if they contain a main method.

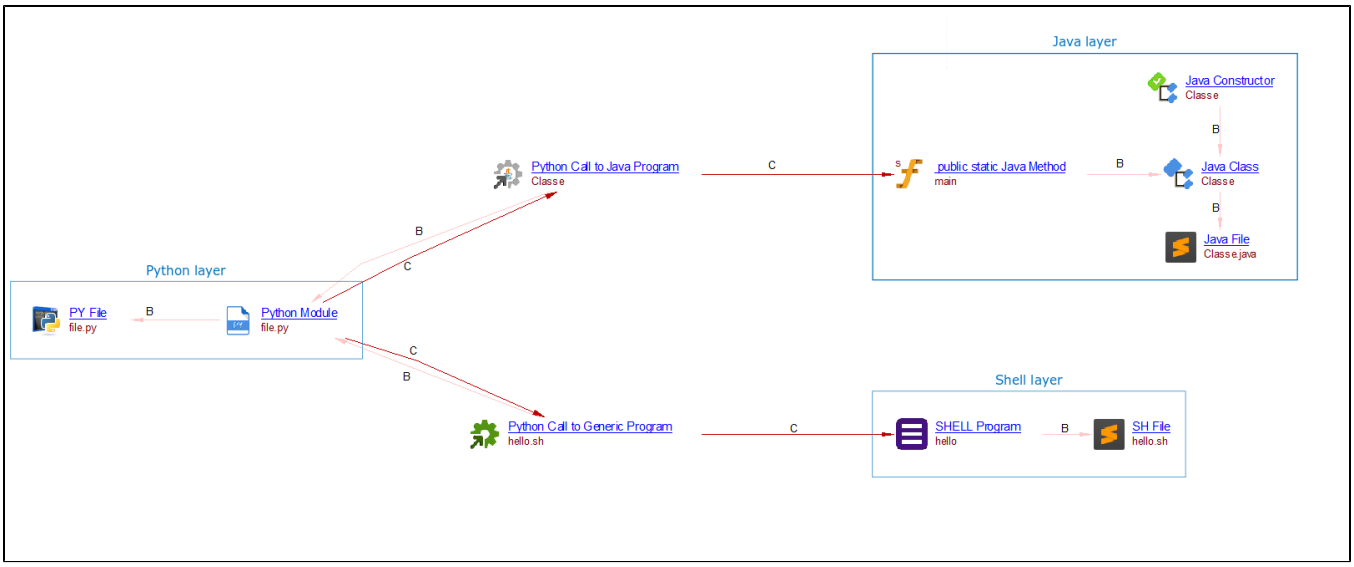
CAST Enlighten screenshots

When a call to an external program is analyzed by the Python analyzer, the following transactions can be found at the end of analysis:

Example of call to an external program

```
import subprocess
from subprocess import Popen

subprocess.call('/bin/java com.cast.Classe')
cmd = './hello.sh'
popen = Popen(cmd)
```

CAST Enlighten screenshot of call to an external program



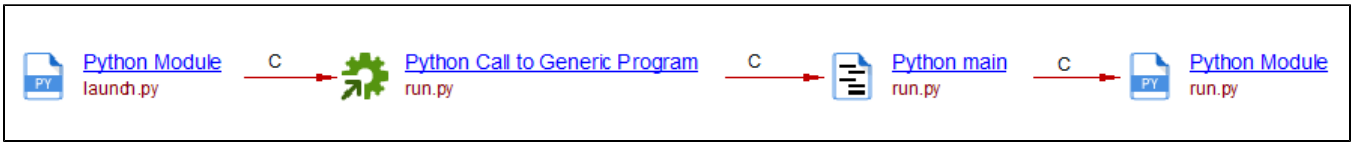
Python code can also call a different Python program via the *python* (or *jython*) executable. Then the analyzer will create, as shown before, "Python Call to Generic Program" objects and they will be linked to the corresponding "Python Main" objects during application level analysis via web service linker extension. For example *launch.py* will invoke the *run.py* script in the code below

```
# launch.py

import subprocess
from subprocess import Popen

cmd = 'python run.py'
popen = Popen(cmd)
```

where the target code contains a code block in the top-level script environment (signaled by the "*if __name__ ...*" structure).

```
# run.py

def run():
    print("running...")

if __name__=="__main__":
    run()
```

so as a results we would have



Links handled by command line parsers

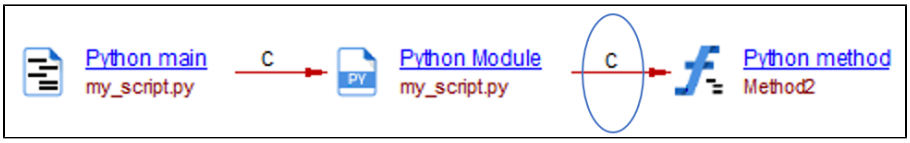
The Python analyzer fulfills the *call-links* handled by the **plac** framework that facilitates the manipulation of command line arguments of Python scripts.

Example of call from plac module

```
class Interface():
    commands = ['Method2']

    def Method1(self):
        pass

    def Method2(self):
        pass

if __name__ == '__main__':
    plac.Interpreter.call(Interface)
```

In this example, the "script" character of the source file is followed by the presence of the "if __name__ == ..." structure. This structure is represented by the analyzer with a *Python main* object that serves as an entry point for receiving (external) calls. The call handled by *plac* between *plac.Interpreter.call()* an d *Method2* will be modelized as call-link by the Python analyzer as shown below.

CAST Enlighten screenshot of call handled by *plac.*



Limitations

- Not fully supported **Python Decorator function**.
- Quality rules do not apply to code inside the class definition (**class** or **"static" variables**)
- The "Avoid disabling certificate check when requesting secured urls" for 'urllib3' is only partially supported by detecting the call to 'urllib3. disable_warnings'.
- Limited Python resolution that leads to missing links:
  - No support for __**all**__
  - No support for variable of type class, function
- Flask:
  - Objects for other web service operations such as PATCH are not generated.
  - The *endpoint* abstraction layer between functions and annotations is not considered. When using *add_url_rule* the endpoint argument is taken as the calling function name.
- **Django** framework is not supported.
- Java-Python interoperability via **Jython** is not supported. However the files with the specific extension .jy for Jython is analyzed as a regular Python file.
- Message queues
  - To generate queue message objects the queue name has to be initialized explicitly in the code (dynamic naming not supported).