Usage Examples

Encoding & Decoding Tokens with HS256

```
>>> import jwt
>>> key = "secret"
>>> encoded = jwt.encode({"some": "payload"}, key, algorithm="HS256")
>>> print(encoded)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCJ9.4twFt5NiznN84AWoo1d7KO1T_yoc0Z6

>>> jwt.decode(encoded, key, algorithms="HS256")
{'some': 'payload'}
```

Encoding & Decoding Tokens with RS256 (RSA)

RSA encoding and decoding require the `cryptography` module. See Cryptographic Dependencies (Optional).

```
>>> import jwt
>>> private_key = b"-----BEGIN PRIVATE KEY-----\nMIGEAgEAMBAGByqGSM49AgEGBS..."
>>> public_key = b"-----BEGIN PUBLIC KEY-----\nMHYwEAYHKoZIzj0CAQYFK4EEAC..."
>>> encoded = jwt.encode({"some": "payload"}, private_key, algorithm=" RS256 ")
>>> print(encoded)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCJ9.4twFt5NiznN84AWoo1d7KO1T_yoc0Z6

>>> decoded = jwt.decode(encoded, public_key, algorithms=[" RS256 "])
{'some': 'payload'}
```

If your private key needs a passphrase, you need to pass in a `PrivateKey` object from `cryptography`.

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend

pem_bytes = b"-----BEGIN PRIVATE KEY-----\nMIGEAgEAMBAGByqGSM49AgEGBS..."
passphrase = b"your password"

private_key = serialization.load_pem_private_key(
    pem_bytes, password=passphrase, backend=default_backend()
)
encoded = jwt.encode({"some": "payload"}, private_key, algorithm=" RS256 ")
```

Specifying Additional Headers

```
>>> jwt.encode(
...     {"some": "payload"},
...     "secret",
...     algorithm="HS256",
...     headers={"kid": "230498151c214b788dd97f22b85410a5"},
... )
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IjIzMDQ5ODE1MWMyMTRiNzg4ZGQ5N2YyMmI4NTQxMGE1In0.
R6geQ2nMU2BRM-LnYEtefwg'
```

Reading the Claimset without Validation

If you wish to read the claimset of a JWT without performing validation of the signature or any of the registered claim names, you can set the `verify_signature` option to `False`.

Note: It is generally ill-advised to use this functionality unless you clearly understand what you are doing. Without digital signature information, the integrity or authenticity of the claimset cannot be trusted.

```
>>> jwt.decode(encoded, options={"verify_signature": False})
{'some': 'payload'}
```

Reading Headers without Validation

Some APIs require you to read a JWT header without validation. For example, in situations where the token issuer uses multiple keys and you have no way of knowing in advance which one of the issuer's public keys or shared secrets to use for validation, the issuer may include an identifier for the key in the header.

```
>>> jwt.get_unverified_header(encoded)
{'alg': ' RS256 ', 'typ': 'JWT', 'kid': 'key-id-12345...'}
```

Registered Claim Names

The JWT specification defines some registered claim names and defines how they should be used. PyJWT supports these registered claim names:

- “exp” (Expiration Time) Claim
- “nbf” (Not Before Time) Claim
- “iss” (Issuer) Claim
- “aud” (Audience) Claim
- “iat” (Issued At) Claim

Expiration Time Claim (exp)

> The “exp” (expiration time) claim identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. The processing of the “exp” claim requires that the current date/time MUST be before the expiration date/time listed in the “exp” claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

You can pass the expiration time as a UTC UNIX timestamp (an int) or as a datetime, which will be converted into an int. For example:

```
jwt.encode({"exp": 1371720939}, "secret")
jwt.encode({"exp": datetime.now(tz=timezone.utc)}, "secret")
```

Expiration time is automatically verified in *jwt.decode()* and raises *jwt.ExpiredSignatureError* if the expiration time is in the past:

```
try:
    jwt.decode("JWT_STRING", "secret", algorithms=["HS256"])
except jwt.ExpiredSignatureError:
    # Signature has expired
    ...
```

Expiration time will be compared to the current UTC time (as given by *timegm(datetime.now(tz=timezone.utc).utctimetuple())*), so be sure to use a UTC timestamp or datetime in encoding.

You can turn off expiration time verification with the *verify_exp* parameter in the options argument.

PyJWT also supports the leeway part of the expiration time definition, which means you can validate a expiration time which is in the past but not very far. For example, if you have a JWT payload with a expiration time set to 30 seconds after creation but you know that sometimes you will process it after 30 seconds, you can set a leeway of 10 seconds in order to have some margin:

```
jwt_payload = jwt.encode(
    {"exp": datetime.datetime.now(tz=timezone.utc) + datetime.timedelta(seconds=30)},
    "secret",
)

time.sleep(32)

# JWT payload is now expired
# But with some leeway, it will still validate
jwt.decode(jwt_payload, "secret", leeway=10, algorithms=["HS256"])
```

Instead of specifying the leeway as a number of seconds, a *datetime.timedelta* instance can be used. The last line in the example above is equivalent to:

```
jwt.decode(
    jwt_payload, "secret", leeway=datetime.timedelta(seconds=10), algorithms=["HS256"]
)
```

Not Before Time Claim (nbf)

> The “nbf” (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. The processing of the “nbf” claim requires that the current date/time MUST be after or equal to the not-before date/time listed in the “nbf” claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

The *nbf* claim works similarly to the *exp* claim above.

```
jwt.encode({"nbf": 1371720939}, "secret")
jwt.encode({"nbf": datetime.now(tz=timezone.utc)}, "secret")
```

Issuer Claim (iss)

> The “iss” (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. The “iss” value is a case-sensitive string containing a StringOrURI value. Use of this claim is OPTIONAL.

```
payload = {"some": "payload", "iss": "urn:foo"}

token = jwt.encode(payload, "secret")
decoded = jwt.decode(token, "secret", issuer="urn:foo", algorithms=["HS256"])
```

If the issuer claim is incorrect, *jwt.InvalidIssuerError* will be raised.

Audience Claim (aud)

> The “aud” (audience) claim identifies the recipients that the JWT is intended for. Each principal intended to process the JWT MUST identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the “aud” claim when this claim is present, then the JWT MUST be rejected.

In the general case, the “aud” value is an array of case- sensitive strings, each containing a StringOrURI value.

```
payload = {"some": "payload", "aud": ["urn:foo", "urn:bar"]}

token = jwt.encode(payload, "secret")
decoded = jwt.decode(token, "secret", audience="urn:foo", algorithms=["HS256"])
```

In the special case when the JWT has one audience, the “aud” value MAY be a single case-sensitive string containing a StringOrURI value.

```
payload = {"some": "payload", "aud": "urn:foo"}

token = jwt.encode(payload, "secret")
decoded = jwt.decode(token, "secret", audience="urn:foo", algorithms=["HS256"])
```

If multiple audiences are accepted, the `audience` parameter for `jwt.decode` can also be an iterable

```
payload = {"some": "payload", "aud": "urn:foo"}

token = jwt.encode(payload, "secret")
decoded = jwt.decode(
    token, "secret", audience=["urn:foo", "urn:bar"], algorithms=["HS256"]
)
```

The interpretation of audience values is generally application specific. Use of this claim is OPTIONAL.

If the audience claim is incorrect, *jwt.InvalidAudienceError* will be raised.

Issued At Claim (iat)

> The iat (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.
>
> If the *iat* claim is not a number, an *jwt.InvalidIssuedAtError* exception will be raised.

```
jwt.encode({"iat": 1371720939}, "secret")
jwt.encode({"iat": datetime.now(tz=timezone.utc)}, "secret")
```

Requiring Presence of Claims

If you wish to require one or more claims to be present in the claimset, you can set the `require` parameter to include these claims.

```
>>> jwt.decode(encoded, options={"require": ["exp", "iss", "sub"]})
{'exp': 1371720939, 'iss': 'urn:foo', 'sub': '25c37522-f148-4cbf-8ee6-c4a9718dd0af'}
```

Retrieve RSA signing keys from a JWKS endpoint

```
>>> import jwt
>>> from jwt import PyJWKClient
>>> token =
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ik5FRTFRVVJCT1RNNE16STVSa0ZETlRZeE9UVTFNRGcyT0Rn
GXvgFzsdsyWlVQvPX2xjeaQ217r2PtxDeqjlf66UYl6oY6AqNS8DH3iryCvIfCcybRZkc_hdy-
6ZMoKT6Piijvk_aXdm7-QQqKJFHLuEqrVSOuBqqiNfVrG27QzAPuPOxvfXTVLXL2jek5meH6n-
VWgrBdoMFH93QEszEDowDAEhQPHVs0xj7SIzA"
>>> kid = "NEE1QURBOTM4MzI5RkFDNTYxOTU1MDg2ODgwQ0UzMTk1QjYyRkRFQw"
>>> url = "https://dev-87evx9ru.auth0.com/.well-known/jwks.json"
>>> jwks_client = PyJWKClient(url)
>>> signing_key = jwks_client.get_signing_key_from_jwt(token)
>>> data = jwt.decode(
...     token,
...     signing_key.key,
...     algorithms=[" RS256 "],
...     audience="https://expenses-api",
...     options={"verify_exp": False},
... )
>>> print(data)
{'iss': 'https://dev-87evx9ru.auth0.com/', 'sub':
'aW4Cca79xReLWUz0aE2H6kD0O3cXBVtC@clients', 'aud': 'https://expenses-api', 'iat':
1572006954, 'exp': 1572006964, 'azp': 'aW4Cca79xReLWUz0aE2H6kD0O3cXBVtC', 'gty': 'client-
credentials'}
```