Chapitre 00

Protocole HTTP et programmation serveur

**ARCHITECTURE CLIENT/SERVEUR**

1. **INTRODUCTION**

Le World Wide Web est le système qui vous permet de naviguer entre les pages en cliquant simplement sur les liens dans un navigateur. Donc, pour résumer, Internet est le réseau, l'infrastructure. Le Web est un service sur ce réseau. L'Internet, l'infrastructure, et le Web, le service permettant de naviguer, sont deux choses très différentes. L'Internet peut exister sans le Web, mais pas l'inverse. Deux choses si différentes qu'elles n'ont pas été créées en même temps. *ARPAnet*, c'est-à-dire l'ancêtre d'Internet, a été créé en 1967, alors que le Web, compris comme un moyen de naviguer à l'aide de liens, date de 1989. Le premier navigateur graphique date de 1992.

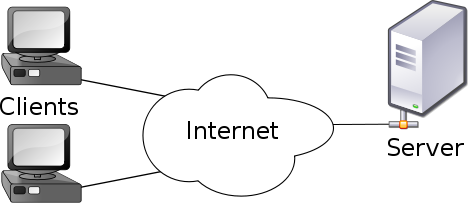
Internet est ensemble de réseaux informatiques locaux utilisant les mêmes protocoles de bas niveau standards (TCP/IP) et formant un réseau global.

Le Web est l’une des applications d’internet, et la plus populaire, d’où l’amalgamme souvent fait entre les deux.

Figure 8 Internet vs web

Ces technologies peuvent bien sûr évoluer (elle l’ont déjà fait), voire être à terme remplacées par d’autres. Ce ne sont pas elles qui définissent le Web, mais les 3 propriétés qu’elles lui confèrent.

* Architecture du Web
* Architecture Client-Serveur

[](https://perso.liris.cnrs.fr/pierre-antoine.champin/2017/progweb-python/_images/client-server.png)

Il est même possible d’avoir, sur un même ordinateur, un logiciel client et un logiciel serveur.

**Rôles du serveur**

Est garant de l’état des ressources,

* Pour en fournir une représentation aux clients qui le demandent (GET),
* Mais aussi pour les modifier en réponse à certaines requêtes (POST), par exemple :
  + Commande sur un site marchand,
  + Message posté sur un réseau social,
* Etc...

L’état d’une ressource n’est pas forcément stocké dans un fichier. Souvent, il est stocké dans une base de données ; la représentation HTML (ou autre) de la ressource est calculée par le serveur.

**Rôles du client**

* Interprète les représentations envoyées par le serveur (pour les afficher, mais pas uniquement)
* Gère la navigation en interne
* Gère les interactions avec le serveur :
  + clics sur des liens,
  + formulaires,
* etc...

**LES APPLICATIONS RÉSEAU**

* Applications = la raison d'être des réseaux infos
* Profusion d'applications depuis 30 ans grâce à l'expansion d'Internet
* années 1980/1990 : les applications "textuelles"
* messagerie électronique, accès à des terminaux distants, transfert de fichiers, groupe de discussion (forum, newsgroup), dialogue interactif en ligne (chat), la navigation Web
* plus récemment :
* les applications multimédias : vidéo à la demande (streaming), visioconférences, radio et téléphonie sur Internet
* la messagerie instantanée (ICQ, MSN Messenger)
* les applications Peer-to-Peer (MP3, …)

Message HTTP

Principe

HTTP est basé sur l’échange de **messages** :

* le client envoie un message requête,
* le serveur retourne un message réponse.

Chaque échange est indépendant des autres (messages auto-suffisants).

**Structure générale**

**Structure d’une requête**

* Première ligne
  + verbe
  + identifiant local de la ressource
  + version du protocole HTTP
* En-têtes (suivis d’une ligne vide)
* Contenu facultatif (selon le verbe)

**Exemples de requêtes**

**GET** /france/lyon **HTTP/**1.1

Host**:** meteo.example.org

User-Agent**:** Mozilla/5.0 (X11; Linux x86\_64;

rv:58.0) Gecko/20100101 Firefox/58.0

Accept**:** text/html,application/xhtml+xml,

application/xml;q=0.9,\*/\*;q=0.8

Accept-Language**:** fr,en;q=0.5

Accept-Encoding**:** gzip,deflate

Accept-Charset**:** UTF-8,\*

Connection**:** keep-alive

Keep-Alive**:** 300

(pas de contenu)

**GET** /france?ville=lyon **HTTP/**1.1

Host**:** meteo.example.org

User-Agent**:** Mozilla/5.0 (X11; Linux x86\_64;

rv:58.0) Gecko/20100101 Firefox/58.0

Accept**:** text/html,application/xhtml+xml,

application/xml;q=0.9,\*/\*;q=0.8

Accept-Language**:** fr,en;q=0.5

Accept-Encoding**:** gzip,deflate

Accept-Charset**:** UTF-8,\*

Connection**:** keep-alive

Keep-Alive**:** 300

(pas de contenu)

**POST** /passer-commande **HTTP/**1.1

Host**:** marchand.example.org

User-Agent**:** Mozilla/5.0 (X11; Linux x86\_64;

rv:58.0) Gecko/20100101 Firefox/58.0

Accept**:** text/html,application/xhtml+xml,

application/xml;q=0.9,\*/\*;q=0.8

Accept-Language**:** fr,en;q=0.5

Accept-Encoding**:** gzip,deflate

Accept-Charset**:** UTF-8,\*

Connection**:** keep-alive

Keep-Alive**:** 300

Content-Type**:** application/x-www-form-urlencoded

Content-Length**:** 12345

nom=PA+Champin&addresse=12+rue+Turing&articles=...

On retrouve dans la requête la structure de l’URL (nom du serveur, nom local de la ressource)

Notez que les deux premières requêtes sont identiques, à l’exception du nom local.

**Structure d’une réponse**

* Première ligne
  + version du protocole HTTP
  + code de statut
  + libellé textuel
* En-têtes (suivis d’une ligne vide)
* Contenu facultatif (selon le code de statut)

### Exemples de réponse

**HTTP/**1.1 200 **OK**

Date**:** Mon, 02 Jan 2016 22:46:26 GMT

Server**:** Apache/2

Accept-Ranges**:** bytes

Content-Type**:** text/html; charset=utf-8

Content-Length**:** 29794

Etag**:** "7462-477341dcfb940;89-3f26bd17a2f00"

Last-Modified**:** Mon, 02 Jan 2016 12:00:00 GMT

Content-Location**:** Home.html

Vary**:** negotiate,accept

Cache-Control**:** max-age=600

Expires**:** Mon, 02 Nov 2009 22:56:26 GMT

Connection**:** close

**<!DOCTYPE html>**

<html><head><title>Météo de Lyon</title>

...

**HTTP/**1.1 303 **See also**

Date**:** Mon, 02 Jan 2016 22:46:26 GMT

Server**:** Apache/2

Accept-Ranges**:** bytes

Location**:** /commande/12345

Connection**:** close

(pas de contenu)

**HTTP/**1.1 404 **Not Found**

Date**:** Mon, 02 Jan 2016 22:46:26 GMT

Server**:** Apache/2

Content-Type**:** text/html; charset=utf-8

Content-Length**:** 2979http://rdflib.readthedocs.io/

Connection**:** close

**<!DOCTYPE html>**

<html><head><title>Cette ressource n'existe pas</title>

...

### Codes de statut

HTTP définit 40 codes de statut, répartis en cinq catégories :

| **Catégories** | **Exemples** |
| --- | --- |
| 1xx : Information | 100 Continue |
| 2xx : Succès | 200 OK |
| 3xx : Redirection | 301 Moved Permanently |
| 4xx : Erreur client | 404 Not Found, 401 Unauthorized |
| 5xx : Erreur serveur | 500 Internal Server Error |

### En-têtes de requête

**Note**

HTTP spécifie un très grand nombre d’en-têtes ; nous décrirons au fur et à mesure du cours ceux dont nous avons besoin.

* accept: la liste des types de contenu (au format [MIME](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_Types)) préférés le client.
* accept-language: la liste des langues (au format [**RFC 4646**](https://tools.ietf.org/html/rfc4646.html)) préférés par le client.

Ces en-têtes servent à la négociation de contenu (conneg). Le serveur peut utiliser ces informations pour adapter le contenu de la réponse, mais il peut aussi les ignorer (lorsque la ressources est un fichier statique).

* cache-control: permet d’influer sur le comportement des caches intermédiaires, notamment pour les inhiber (option no-cache).

### En-têtes de réponse

* content-type: le type de contenu de la réponse (au format [MIME](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_Types)). Exemple: text/html, image/png, application/pdf.
* content-length: la taille du message en octets.
* cache-control: permet d’influer sur le comportement des caches, notamment pour indiquer la durée de validité maximale (option max-age=<seconds>).

**Note**

les en-têtes content-type et content-length sont en fait utilisés pour tout message ayant un contenu, y compris certaines requêtes (*e.g.* POST).

## WSGI

### Serveur et application

digraph {
  rankdir=LR;
  edge [arrowTail=normal; dir=both];
  client -> serveur
  serveur -> fichier
  serveur -> PHP
  serveur -> "programme CGI"
  serveur -> "python (WSGI)"
  serveur -> "..."
}

*Les ressources du serveur peuvent être gérées de différentes manières*

**Note**

* Certaines ressources sont stockées directement dans des fichiers statiques (dont le nom est en général corellé avec le chemin de l’URL).
* Certaines ressources sont gérées par un script PHP.
* Certaines ressources sont gérées par un programme, répondant à certains standards, comme
  + [CGI](https://en.wikipedia.org/wiki/Common_Gateway_Interface),
  + WSGI, que nous allons décrire dans ce cours,
  + ...

### WSGI

WSGI (Web Server Gateway Interface) est un standard spécifiant comment un serveur Web peut interagir avec une application Python.

### Hello world WSGI

**def** **application**(environ, start\_response):

message **=** b"Hello world\n"

status **=** "200 Ok"

headers **=** [

("content-type", "text/html"),

("content-length", str(len(message))),

]

start\_response(status, headers)

**return** [message]

### Explications

* La fonction application sera appelée à chaque requête.
* environ est un dictionnare, qui contient notamment toute l’information décrivant la requête.
* start\_response est une fonction servant à générer les méta-données de la réponse (statut et en-têtes).
* La fonction doit retourner une liste de chaînes d’octets.

**Note**

Plus précisément, la fonction doit retourner un [itérable](https://docs.python.org/3/glossary.html#term-iterable) de chaînes d’octets. Cela permet dans certains cas au serveur d’envoyer les premiers octers avant que la fonction n’ait terminé son exécution.

### Chargement du module

* Le module peut rester chargé entre deux requêtes, ce qui permet de mutualiser certains traitemens (*e.g.* connexion à une base de données).
* Cela dit, le serveur peut à tout moment décharger le module, donc l’application ne doit pas s’appuyer sur des données conservées en mémoire.

n **=** 0

**def** **application**(environ, start\_response):

**global** n

n **=** n**+**1

message **=** ("Compteur: %s\n" **%** n)**.**encode("utf-8")

status **=** "200 Ok"

headers **=** [ ("content-type", "text/html"),

("content-length", str(len(message))), ]

start\_response(status, headers)

**return** [message]

**Note**

Dans l’exemple ci-dessus, plusieurs requêtes successives vont faire augmenter le compteur. Cependant, à tout moment, le module peut être déchargé par le serveur, et rechargé lors d’une nouvelle requête, ce qui remettra le compteur à zéro. Si on voulait persister la valeur du compteur, il faudrait la sauvegarder dans un fichier ou une base de données.

### Le dictionnaire environ

Il contient, entre autre, les clé suivantes :

* PATH\_INFO: la partie hiérarchique du nom de la ressource
* QUERY\_STRING: la partie associative du nom de la ressource
* REQUEST\_METHOD: la méthode HTTP de la requête
* wsgi.input: un file-like donnant accès au contenu de la requête
* CONTENT\_TYPE, CONTENT\_LENGTH: la valeur des en-têtes correspondants
* HTTP\_xxx : la valeur le l’en-tête xxx

Pour en savoir plus, consultez [la spécification WSGI](https://www.python.org/dev/peps/pep-3333/#environ-variables).

### Serveur de développement

* Pour tester notre application, il ne serait pas très pratique d’avoir à configurer et déployer un serveur Web complet.
* Heureusement, les bibliothèques standards de Python nous fournissent un mini-serveur, utile pour la phase de développement.
* Pour l’utiliser, il suffit de créer un script contenant les lignes suivantes (en adaptant la première ligne) :

**from** mon\_projet **import** application

**from** wsgiref.simple\_server **import** make\_server

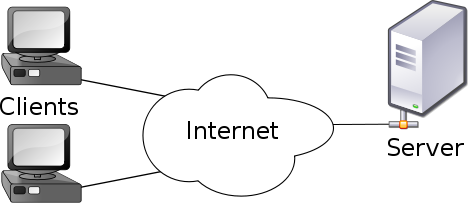
srv **=** make\_server("localhost", 12345, application)

srv**.**serve\_forever()

# API Web

## Motivation

### Arichitecture client-serveur

[](https://perso.liris.cnrs.fr/pierre-antoine.champin/2017/progweb-python/_images/client-server.png)

*Source image*[*http://commons.wikimedia.org/wiki/File:Client-server-model.svg*](http://commons.wikimedia.org/wiki/File:Client-server-model.svg)

### Hétérogénéité des clients

* client mobile
  + minimiser les échanges (bande passante, batterie, forfait...)
  + gérer l’affichage coté client
* autre serveur
  + exemple : agence de voyage / SNCF
  + pas besoin d’affichage

**Note**

Même dans le cas d’une application HTML “classique”, on peut bénéficier d’une API grâce à [AJAX](https://perso.liris.cnrs.fr/pierre-antoine.champin/2017/progweb-python/cours/cm5.html).

### Définition

* API = *Application Programming Interface*
* Permet donc de programmer une application s’appuyant sur le service Web
* (par opposition à une utilisation “manuelle” à travers un navigateur).

### Comment tester une API ?

* Sous Firefox : [RestClient](https://addons.mozilla.org/fr/firefox/addon/restclient/)
* Sous Chrome : [PostMan](https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop)

## Le format JSON

### Présentation

* JavaScript Object Notation (JSON) est un format largement utilisé pour échanger des données sur le Web.
* Inspiré du langage Javascript, il est en également bien adapté à de nombreux langages, dont Python.

### Sémantique

JSON permet de représenter les données Python suivantes :

* None,
* des chaînes de caractères,
* des nombres (entiers et flottants),
* des booléens,
* des listes,
* des dictionnaires dont les clés sont des chaînes de caractères.

### Exemple de données représentables en JSON

{

'id': 1,

'label': "File",

'tooltip': None,

'items': [

{'label': "New", 'visible': True},

{'label': "Open", 'visible': True},

{'label': "Close", 'visible': False},

],

}

### Syntaxe

Très similaire à Python, à quelques exceptions près :

* None s’écrit null,
* True s’écrit true,
* False s’écrit false,
* les chaînes de caractères sont obligatoirement encadrées par des *double quotes*.

**Note**

Ces variantes proviennent principalement du langage Javascript, mais aussi d’une volonté de garder le langage JSON simple.

En terme de vocabulaire,

* on parle de tableau (*array*) plutôt que de liste,
* on parle d’objet plutôt que de dictionnaire.

### Exemple précédent dans la syntaxe JSON

{

"id": 1,

"label": "File",

"tooltip": **null**,

"items": [

{"label": "New", "visible": **true**},

{"label": "Open", "visible": **true**},

{"label": "Close", "visible": **false**}

]

}

### Utilisation en python

En Python standard :

**import** json

*# avec des fichiers*

data **=** json**.**load(open("file.json"))

json**.**dump(data, open("file2.json", "w"))

*# avec des chaînes de caractères*

txt **=** json**.**dumps(data)

data2 **=** json**.**loads(txt)

Dans Flask:

* request.get\_json() fournit le contenu JSON de la requête (le cas échéant), ou None si le contenu est absent ou dans autre format.
* flask.jsonify(data) produit un objet réponse dont
  + l’en-tête content-type est application/json, et
  + le contenu est la sérialisation JSON des données passées en paramètre.

## Verbes HTTP

### Introduction

Avec HTML, on utilise exclusivement les verbes GET et POST, mais HTTP définit d’autres verbes, qui sont particulièrement utiles lors de la définition d’APIs.

### GET

* n’attend pas contenu
* réclame une représentation de la ressource
* est **sans effet de bord** (*i.e.* ne modifie pas l’état de la ressource)
* rend possible l’utilisation des caches

### HEAD

* a la même sémantique que GET
* mais ne réclame que les en-têtes
* et pas la représentation de la ressource

### PUT

* attend un contenu
* demande la création ou la modification de la ressource
* de sorte que le nouvel état de la ressource corresponde au contenu fourni
* a donc un effet de bord **idempotent**

### DELETE

* n’attend pas de contenu
* demande la suppression de la ressource
* a donc un effet de bord **idempotent**

### POST

* attent un contenu
* peut avoir n’importe quel effet de bord (y compris non-idempotent)
* est souvent utiliser pour créer une ressource sans connaître a priori son URL

### API RESTful

* On affecte à chaque ressource une URL qui l’identifie,
* et chaque URL identifie une ressource.
* Les manipulations se font en utilisant les verbes HTTP appropriés.

**Avertissement**

Dans le [TP précédent](https://perso.liris.cnrs.fr/pierre-antoine.champin/2017/progweb-python/tdp/tdp3.html), nous n’avons pas tout à fait respecté ces principes... (exemple : quelle ressource identifie l’URL /Genes/del/<id> ?)

**Note**

L’acronyme REST et ses dérivés (comme “RESTful”) a été inventé par Roy Fieldings dans [sa thèse](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm).

### Patron “collection/élément”

* une collection contient une liste d’éléments
* la collection supporte
  + GET pour obtenir la liste de ses éléments (avec leurs URLs)
  + POST pour créer un nouvel élément
* chaque élément supporte
  + GET pour en obtenir une description détaillée
  + PUT pour le modifier
  + DELETE pour le supprimer

### Mise en œuvre en Flask

Il est possible d’implémenter différents verbes dans la même vue :

@app**.**route("/bidules/", methods**=**['GET', 'POST'])

**def** **bidules**():

**if** request**.**method **==** 'GET':

*# ...*

**else**:

*# ...*

Mais il est également possible de les implémenter dans différentes vues, (et donc d’avoir plusieurs routes ayant la même URL, mais des verbes différents) :

@app**.**route("/bidules/", methods**=**['GET'])

**def** **bidules\_list**():

*# ...*

@app**.**route("/bidules/", methods**=**['POST'])

**def** **bidules\_new**():

*# ...*

Le choix entre les deux options dépend de la quantité de code commun aux traitements des différents verbes.

## Gestion du cache

### Introduction

* HTTP est conçu pour tirer parti des caches.
* Cette gestion de cache n’est **pas spécifique** au développement d’APIs, elle est également utile pour tous les fichiers réutilisés dans plusieurs pages (images, feuilles de styles, *etc*).
* Mais elle est d’autant plus importante avec les APIs : une machine peut lancer des requêtes beaucoup plus vite qu’un humain.

### En-tête Cache-control

Voici deux valeurs typiques pour cache-control dans une réponse :

* max-age=T (ou T est une durée en secondes) : informe le client sur la durée pendant laquelle il est pertinent de garder l’information en cache.
* no-cache informe le client qu’il ne doit pas conserver cette réponse en cache, pour le forcer à ré-interroger le serveur à la prochaine requête.

### En-tête ETag

* Un **etag** (*entity tag*) est une chaîne de caractère identifiant une version de la ressource. L’etag change chaque fois que la ressource est modifiée.
* Elle permet au client d’indiquer au serveur la dernière version de la ressource dont il a connaissance.
* Ceci ce fait en conditionnant l’exécution d’une requête à certaines versions de la ressource, avec les en-têtes
  + If-Match: seulement si la ressource correspond à l’un des etags fournis
  + If-None-Match: seulement si la ressource ne correspond à aucun des etags fournis

### Etag et GET

ClientServerReusescachedversionGET /foo200 OKEtag: "abc"GET /fooIf-none-match: "abc"304 Not ModifiedGET /fooIf-none-match: "abc"200 OKEtag: "cde"Resource changes

### Etag et PUT - Concurrence optimiste

Client AServerClient BResourcechangedGET /foo200 OKEtag: "abc"GET /foo200 OKEtag: "abc"PUT /fooIf-match: "abc"200 OKEtag: "def"PUT /fooIf-match: "abc"409 ConflictGET /foo200 OKEtag: "def"

### Mise en œuvre en Flask

* L’objet Response a une méthode set\_etag qui permet de lui associer un etag.

**Note**

Il ne faut pas utiliser resp.headers["etag"] = my\_etag car les etags doivent être encodés d’une manière spécifique.

* L’objet request a des attributs if\_match et if\_none\_match qui contiennent la liste des etags fournis par le client (correctement décodés).
* Exemple d’utilisation :
* @app**.**route('/my/url')
* **def** **smart\_view**():
* etag **=** get\_etag()
* **if** etag **in** request**.**if\_none\_match:
* resp **=** make\_response("", 304) *# 304 not modified*
* resp**.**set\_etag(etag)
* **return** resp
* *# here, make the "normal" response*

**Note**

Flask offre une manière plus intégrée de gérer les requêtes GET conditionnelles :

@app**.**route('/my/url')

**def** **smart\_view**():

*# here, make the "normal" response*

resp**.**set\_etag(get\_etag())

resp**.**make\_conditional(request)

*# transform the response according to conditional request*

**return** resp

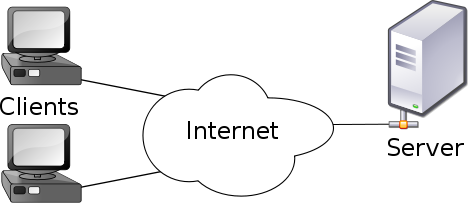
La réponse sera modifiée pour tenir compte de l’en-tête If-None-March, mais également de l’en-tête Range ([**RFC 7233**](https://tools.ietf.org/html/rfc7233.html)). Cette solution est donc plus concise et plus puissante, mais elle oblige à construire systématiquement la réponse, ce qui peut avoir un coût significatif (accès à la base de données).

Le choix de l’une ou l’autre solution est donc une question de compromis.

# Pages dynamiques en Javascript

## Motivation

### Architecture Client-Serveur

[](https://perso.liris.cnrs.fr/pierre-antoine.champin/2017/progweb-python/_images/client-server.png)

*Source image*[*http://commons.wikimedia.org/wiki/File:Client-server-model.svg*](http://commons.wikimedia.org/wiki/File:Client-server-model.svg)

* Jusqu’à maintenant, le gros du travail était fait par le serveur.
* On souhaite pouvoir déporter une partie de la **logique applicative** coté client.

### Programmation coté client

* Ceci suppose d’avoir un langage de programmation généraliste (≠ HTML/CSS) compris par tous les navigateurs.
* Actuellement, ce langage est Javascript.

**Note**

Javascript a beaucoup évolué au cours de son histoire.

La version que nous présentons ici est ES6, une version relativement récente (2015) qui a apporté beaucoup de nouveautés au langage, mais également des incompatibilités avec les versions précédentes.

**Gardez cela en tête lorsque vous trouverez des exemples**

en ligne.

## Syntaxe

### Inspiration

* Comme son nom l’indique, la syntaxe de Javascript est (librement) inspirirée de celle de Java (ou du C).
* La similitude s’arrête là : Javascript n’est pas basé sur Java.

### Condition

**if** (i **<** 10) {

j **=** j**+**1;

k **+=** i;

} **else** {

j **=** 0;

}

**if** i **<** 10:

j **=** j**+**1

k **+=** i

**else**:

j **=** 0

### Boucles

**while** (i **<** 10) {

j **=** j**\***i;

i **+=** 1;

}

**while** i **<** 10:

j **=** j**\***i

i **+=** 1

**for**(**let** i **of** [1,1,2,3,5,8]) {

j **=** j**\***i;

}

**for** i **in** [1,1,2,3,5,8]:

j **=** j**\***i

**for**(**let** i**=**2; i**<**1000; i**=**i**\***i) {

console.log(i);

}

i **=** 2

**while** i**<**1000:

**print**(i)

i **=** i**\***i

### Fonctions

**function** fact(n) {

**let** f **=** 1;

**while** (n**>**1) {

f **=** f**\***n;

n **-=** 1;

}

**return** f;

}

**def** **fact**(n):

f **=** 1

**while** n**>**1:

f **=** f**\***n

n **-=** 1

**return** f

### Exceptions

**if** (i **<** 0) {

**throw** **new** Error(

"negative value");

}

**if** i **<** 0:

**raise** **Exception**(

"negative value")

**try** {

i **=** riskyFunction();

}

**catch** (err) {

i **=** **-**1;

}

**try**:

i **=** riskyFunction()

**except** **Exception** **as** err:

i **=** **-**1

### Tableaux

**let** a **=** [4,1,3,6,4];

**let** i **=** 1;

**while** (i**<**a.length) {

a[i] **=** a[i]**+**a[i**-**1];

i **+=** 1;

}

a **=** [4,1,3,6,4]

i **=** 1

**while** i **<** len(a):

a[i] **=** a[i]**+**a[i**-**1]

i **+=** 1

**Avertissement**

Contrairement à Python, un tableau vide (de longueur 0) est équivalent à true dans une condition.

### Objets / Dictionnaires

**Note**

En Javascript, ce qu’on appelle “objet” se rapproche plus des dictionnaires de Python.

**let** p **=** {

"nom"**:** "Doe",

prénom**:** "John",

};

console.log(p["prénom"])

console.log(p.prénom)

p **=** {

"nom": "Doe",

"prénom": "John",

}

**print**(p["prénom"])

### Pièges

* Javascript est un langage souvent décrié.



*Photo by*[*Eliot Ball*](https://softwareengineering.stackexchange.com/users/62355/eliot-ball)

* De fait, il comporte de nombreux pièges.
* Une bonne manière de les éviter est d’utiliser systématiquement un *linter* tel que [JSHint](http://jshint.com/) (intégré à de nombreux éditeurs).

### Déclaration des variables locales

Contrairement à Python (et à d’autres langages dynamiquement typés), Javascript demande que les variables locales soient déclarées, avec le mot-clé let.

**Indice**

Les paramètres des fonctions font exception à cette règle, puisqu’ils sont déclarés par leur présence dans l’en-tête de la fonction.

**function** fact(n) {

**let** f **=** 1;

**for** (**let** i**=**2; i**<=**n; i**+=**1) {

f **=** f**\***i;

}

**return** f;

}

**Avertissement**

Un oubli du mot-clé let est dangereux, car

* il ne constitue pas une erreur,
* mais dans ce cas la variable est considérée comme globale, ce qui peut créer des bugs difficiles à détecter, comme le montre [cet exemple](http://pythontutor.com/javascript.html#code=//%20exemple%20des%20m%C3%A9faits%20des%20variables%20globales%0Avar%20i%3B%20//%20n%C3%A9cessaire%20pour%20pythontutor,%20qui%20refuse%20les%20variables%20globales%20implicites%0A%0Afunction%20fact%28n%29%20%7B%0A%20%20var%20r%20%3D%201%3B%0A%20%20/).

**Note**

[JSHint](http://jshint.com/) détecte ce type d’erreur dans la plupart des cas, à condition d’activer le contrôle ainsi :

*// jshint undef:true*

### Tests d’égalité

En JS, on teste l’égalité avec l’opérateur ===, et l’inégalité avec l’opérateur !== :

**if** (i **===** j **&&** i **!==** k) *// ...*

**Avertissement**

Les opérateurs habituels == et != existent aussi, mais ils ont une sémantique très inhabituelle, et sont donc généralement évités.

**Note**

En fait, l’opérateur == considère que deux valeurs de types différents sont égales si on peut passer de l’une à l’autre par conversion de type.

Par exemple :

"42" **==** 42 *// est vrai*

C’est un problème, car cela conduit à des choses contre-intuitives :

**if** (a **==** b **&&** b **==** c) {

a **==** c; *// peut être faux*

*// par exemple: a = "0", b = 0, c = "" ;*

*// en effet, 0 et "" sont tous deux équivalents à false*

}

ou encore :

**if** (a **==** b) {

a**+**1 **==** b**+**1; *// peut être faux*

*// par exemple: a = "42" (a+1 == "421") et b = 42 (b+1 == 43)*

}

JSHint signale toute utilisation de == ou !=.

Si l’option n’est pas activée par défaut :

*// jshint:: eqeqeq:true*

### null et undefined

* En Python, il n’existe qu’une seule valeur « nulle » (None).
* En JS, il en existe deux : null et undefined.
* Elles sont utilisées dans des contextes un peu différent ; pour l’instant, retenez surtout que les deux existent.

**Indice**

Les deux sont équivalentes à false dans une condition.

### Éléments inexistants

En Javascript, l’accès à un élément inexistant dans un tableau ou un objet ne déclenche pas d’erreur, mais retourne simplement undefined.

Ceci cause en général une erreur ailleurs dans le code, ce qui rend plus difficile le débogage :

**let** p **=** { "prénom"**:** "John" };

**let** n **=** p.nom; *// n reçoit undefined, pas d'erreur*

**if** (n.length **>** 32) { *// erreur: undefined n'a pas de longueur*

*// ...*

}

## L’arbre DOM

### Présentation

La structure d’un fichier HTML peut être vue comme un arbre.

graph  {

  node [ shape=box, style=rounded ]

  html -- head
   head -- title -- title_txt
  html -- body
   body -- h1 -- h1_txt
   body -- p
    p -- p_txt
    p -- a -- a_txt
   body -- img

  a         [ label="a\nhref='./link'" ]
  img       [ label="img\nsrc='./pic'" ]
  title_txt [ shape=box, style=filled, label="Le titre" ]
  h1_txt    [ shape=box, style=filled, label="Le titre" ]
  p_txt     [ shape=box, style=filled, label="Bonjour le " ]
  a_txt     [ shape=box, style=filled, label="monde" ]

}

### Terminologie

* Cet arbre s’appelle l’arbre DOM (pour *Document Object Model*).
* Les nœuds correspondant aux balises sont appelés des **éléments**.
* Les nœuds contenant le contenu textuels sont simplement appelés des « nœuds texte ».

**Note**

Il existe d’autres types de nœuds (par exemple les nœuds commentaire), mais ils sont plus rarement utiles.

### L’objet document

En JS, la variable globale document contient un objet représentant le document HTML. Elle permet d’accéder aux éléments du document :

* [document.getElementById](http://devdocs.io/dom/document/getelementbyid),
* [document.getElementsByTagName](http://devdocs.io/dom/document/getelementsbytagname),
* [document.getElementsByClassName](http://devdocs.io/dom/document/getelementsbyclassname),
* [document.querySelector](http://devdocs.io/dom/document/queryselector),
* [document.querySelectorAll](http://devdocs.io/dom/document/queryselectorall)

**Note**

Il est plus efficace d’utiliser les méthodes getElement\* que d’utiliser querySelector\* avec les sélecteurs correspondants (tagname ou #identifier).

### Attributs et méthodes d’un élément

**textContent:**

permet de consulter et modifier le contenu textuel de l’élément

**style:**

permet de consulter et modifier l’attribut style de l’élément, sous forme d’un objet ayant un attribut pour chaque propriété CSS. (*e.g.* e.style.fontSize pour la propriété font-size)

**Note**

Pour faciliter l’utilisation en Javascript, la typographie des attributs de style n’est pas la même que celle des propriétés CSS correspondantes. Les tirets (-) sont remplacés par une mise en majuscule de la letter suivante ([CamelCase](https://en.wikipedia.org/wiki/CamelCase)).

**classList:**

permet de consulter et modifier l’attribut class de l’élément, grâce aux méthodes suivantes :

* add(cls): ajoute la classe cls a l’élément.
* remove(cls): retire la classe cls a l’élément.
* contains(cls): indique si l’élément possède actuellement la classe cls.
* toggle(cls): inverse l’état de la classe cls (présente/absente) sur l’élément.

**Note**

Comme en HTML+CSS, il est préférable de spécifier la mise en forme à l’aide de classes dans le CSS, et de modifier ces classes dans le code Javascript, plutôt que la spécifier directement dans le code Javascript à travers l’attribute style.

Les éléments possèdent de nombreux autres attributs; en particulier, chaque attribut HTML a une contrepartie en Javascript.

On peut notamment citer :

* href (pour les <a>)
* src (pour les <img>)
* value (pour les <input>)
* disabled (pour tous les éléments de formulaire)
* checked (pour les cases à cocher)
* *etc*...

Expérimentez sur [cet exemple](http://champin.net/enseignement/intro-js/_static/exemples/element_manipulation.html).

### Parcours du DOM

Récupérer des nœuds depuis un élément e :

* [e.getElementsByTagName](http://devdocs.io/dom/element/getelementsbytagname),
* [e.getElementsByClassName](http://devdocs.io/dom/element/getelementsbyclassname),
* [e.querySelector](http://devdocs.io/dom/element/queryselector),
* [e.querySelectorAll](http://devdocs.io/dom/element/queryselectorall),
* [e.childNodes](http://devdocs.io/dom/node/childnodes),
* [e.children](http://devdocs.io/dom/parentnode/children),
* [e.parentNode](http://devdocs.io/dom/node/parentnode),

### Modification du DOM

Création d’un nœud :

* [document.createElement](http://devdocs.io/dom/document/createelement),
* [document.createTextNode](http://devdocs.io/dom/document/createtextnode),
* [n.cloneNode](http://devdocs.io/dom/node/clonenode)

Une fois créé, le nœud est encore hors de l’arborescence du document (et donc, non affiché). Il est nécessaire de le rattacher à un nœud parent par l’une des méthodes suivante :

* [n.insertBefore](http://devdocs.io/dom/node/insertbefore),
* [n.replaceChild](http://devdocs.io/dom/node/replacechild),
* [n.removeChild](http://devdocs.io/dom/node/removechild),
* [n.appendChild](http://devdocs.io/dom/node/appendchild)

## Intégration JS dans HTML

### Avertissement

Il existe de nombreuses méthodes.

Celle proposée ici vise à être simple et évolutive, mais suppose un navigateur moderne.

### Intégration d’un script à une page

On include dans le HTML (dans le head ou le body) une balise script ayant la structure suivante :

<script src**=**"url\_du\_script.js" defer></script>

Le script sera exécuté après le chargement complet du code HTML.

### Programmation événementielle

* En programmation impérative classique, la fonction principale (main) décrit dans quel ordre les différentes fonctions du programme doivent s’exécuter.
* En programmation événementielle, on « abonne » chaque fonction à un (ou plusieurs) **événement(s)**.
* La fonction s’exécute lorsqu’un événement auquel elle est abonnée se produit.
* Les événement sont (souvent) liés aux interactions de l’utilisateur avec l’application.

### Quelques événements utiles

* [click](http://devdocs.io/dom_events/click)
* [mouseover](http://devdocs.io/dom_events/mouseover)
* [keypressed](http://devdocs.io/dom_events/keypressed)
* [input](http://devdocs.io/dom_events/input)
* [change](http://devdocs.io/dom_events/change)
* [submit](http://devdocs.io/dom_events/submit)

Des listes plus exhaustives sont disponibles [ici](http://www.w3schools.com/tags/ref_eventattributes.asp) et [là](http://devdocs.io/dom_events/).

**Note**

Souvent, et pour des raisons historiques, les noms des événements sont préfixés par on. Attention : le véritable nom de l’événement n’inclut pas ce préfixe.

### Mise en œuvre

**function** incrementCounter() {

**let** i **=** document.getElementsByTagName("input")[0];

i.value **=** Number(i.value) **+** 1;

}

**let** b **=** document.querySelector("button");

b.addEventListener('click', incrementCounter);

<http://jsbin.com/mexuna/1/edit?html,js,output>

**Note**

* La méthode addEventListener associe un comportement (fonction) à un événement émis par un élément.
* ⚠ Attention : le deuxième paramètre de addEventListener est le **nom** d’une fonction, **sans** parenthèses (ce n’est pas un appel de la fonction).

### Autre exemple

**function** incrementCounter() {

**let** b **=** document.getElementsByTagName("body")[0];

**let** p **=** document.createElement("p");

p.textContent **=** "Vous avez cliqué";

p.**class**.add("message");

b.appendChild(p);

}

**let** b **=** document.querySelector("button");

b.addEventListener('click', incrementCounter);