# API Reference



## Commands

- `pyppeteer-install`: Download and install chromium for pyppeteer.

## Environment Variables

- `$PYPPETEER_HOME`: Specify the directory to be used by pyppeteer. Pyppeteer uses this directory for extracting downloaded Chromium, and for making temporary user data directory. Default location depends on platform:
  - Windows: `C:\Users\<username>\AppData\Local\pyppeteer`
  - OS X: `/Users/<username>/Library/Application Support/pyppeteer`
  - Linux: `/home/<username>/.local/share/pyppeteer`
    - or in `$XDG_DATA_HOME/pyppeteer` if `$XDG_DATA_HOME` is defined.

  Details see appdirs's `user_data_dir`.

- `$PYPPETEER_DOWNLOAD_HOST`: Overwrite host part of URL that is used to download Chromium. Defaults to `https://storage.googleapis.com`.

- `$PYPPETEER_CHROMIUM_REVISION`: Specify a certain version of chromium you'd like pyppeteer to use. Default value can be checked by `pyppeteer.__chromium_revision__`.

## Launcher

`pyppeteer.launcher.`**`launch`**(*options: dict = None*, ***kwargs*) → pyppeteer.browser.Browser [source]

Start chrome process and return **`Browser`**.

This function is a shortcut to **`Launcher(options, **kwargs).launch()`**.

Available options are:

- `ignoreHTTPSErrors` (bool): Whether to ignore HTTPS errors. Defaults to `False`.
- `headless` (bool): Whether to run browser in headless mode. Defaults to `True` unless `appMode` or `devtools` options is `True`.
- `executablePath` (str): Path to a Chromium or Chrome executable to run instead of default bundled Chromium.
- `slowMo` (int | float): Slow down pyppeteer operations by the specified amount of milliseconds.
- `args` (List[str]): Additional arguments (flags) to pass to the browser process.
- `ignoreDefaultArgs` (bool): Do not use pyppeteer's default args. This is dangerous option; use with care.
- `handleSIGINT` (bool): Close the browser process on Ctrl+C. Defaults to `True`.
- `handleSIGTERM` (bool): Close the browser process on SIGTERM. Defaults to `True`.
- `handleSIGHUP` (bool): Close the browser process on SIGHUP. Defaults to `True`.
- `dumpio` (bool): Whether to pipe the browser process stdout and stderr into `process.stdout` and `process.stderr`. Defaults to `False`.
- `userDataDir` (str): Path to a user data directory.

- `env` (dict): Specify environment variables that will be visible to the browser. Defaults to same as python process.
- `devtools` (bool): Whether to auto-open a DevTools panel for each tab. If this option is `True`, the `headless` option will be set `False`.
- `logLevel` (int | str): Log level to print logs. Defaults to same as the root logger.
- `autoClose` (bool): Automatically close browser process when script completed. Defaults to `True`.
- `loop` (asyncio.AbstractEventLoop): Event loop (**experimental**).
- `appMode` (bool): Deprecated.

> Note:
> Pyppeteer can also be used to control the Chrome browser, but it works best with the version of Chromium it is bundled with. There is no guarantee it will work with any other version. Use `executablePath` option with extreme caution.

`pyppeteer.launcher.`**`connect`**(*options: dict = None, **kwargs*) → pyppeteer.browser.Browser [source]

Connect to the existing chrome.

`browserWSEndpoint` option is necessary to connect to the chrome. The format is `ws://${host}:${port}/devtools/browser/<id>`. This value can get by **`wsEndpoint`**.

Available options are:

- `browserWSEndpoint` (str): A browser websocket endpoint to connect to. (**required**)
- `ignoreHTTPSErrors` (bool): Whether to ignore HTTPS errors. Defaults to `False`.
- `slowMo` (int | float): Slow down pyppeteer's by the specified amount of milliseconds.
- `logLevel` (int | str): Log level to print logs. Defaults to same as the root logger.
- `loop` (asyncio.AbstractEventLoop): Event loop (**experimental**).

`pyppeteer.launcher.`**`executablePath`**() → str [source]

Get executable path of default chrome.

# Browser Class

*class* `pyppeteer.browser.`**`Browser`**(*connection: pyppeteer.connection.Connection, contextIds: List[str], ignoreHTTPSErrors: bool, setDefaultViewport: bool, process: Optional[subprocess.Popen] = None, closeCallback: Callable[[], Awaitable[None]] = None, **kwargs*) [source]

Bases: **`pyee.EventEmitter`**

Browser class.

A Browser object is created when pyppeteer connects to chrome, either through **`launch()`** or **`connect()`**.

**`browserContexts`**

Return a list of all open browser contexts.

In a newly created browser, this will return a single instance of `[BrowserContext]`

*coroutine* **`close`**() → None [source]

Close connections and terminate browser process.

*coroutine* **createIncogniteBrowserContext**() → pyppeteer.browser.BrowserContext [source]

[Deprecated] Miss spelled method.

Use **createIncognitoBrowserContext()** method instead.

*coroutine* **createIncognitoBrowserContext**() → pyppeteer.browser.BrowserContext [source]

Create a new incognito browser context.

This won't share cookies/cache with other browser contexts.

```
browser = await launch()
# Create a new incognito browser context.
context = await browser.createIncognitoBrowserContext()
# Create a new page in a pristine context.
page = await context.newPage()
# Do stuff
await page.goto('https://example.com')
...
```

*coroutine* **disconnect**() → None [source]

Disconnect browser.

*coroutine* **newPage**() → pyppeteer.page.Page [source]

Make new page on this browser and return its object.

*coroutine* **pages**() → List[pyppeteer.page.Page] [source]

Get all pages of this browser.

Non visible pages, such as `"background_page"`, will not be listed here. You can find then using **pyppeteer.target.Target.page()**.

**process**

Return process of this browser.

If browser instance is created by **pyppeteer.launcher.connect()**, return `None`.

**targets**() → List[pyppeteer.target.Target] [source]

Get a list of all active targets inside the browser.

In case of multiple browser contexts, the method will return a list with all the targets in all browser contexts.

*coroutine* **userAgent**() → str [source]

Return browser's original user agent.

> Note:
> Pages can override browser user agent with **pyppeteer.page.Page.setUserAgent()**.

*coroutine* **version**() → str [source]

Get version of the browser.

**wsEndpoint**

Return websocket end point url.

# BrowserContext Class

*class* pyppeteer.browser.**BrowserContext**(*browser: pyppeteer.browser.Browser, contextId: Optional[str]*) [source]

Bases: **pyee.EventEmitter**

BrowserContext provides multiple independent browser sessions.

When a browser is launched, it has a single BrowserContext used by default. The method **browser.newPage()** creates a page in the default browser context.

If a page opens another page, e.g. with a window.open call, the popup will belong to the parent page's browser context.

Pyppeteer allows creation of "incognito" browser context with browser.createIncognitoBrowserContext() method. "incognito" browser contexts don't write any browser data to disk.

```
# Create new incognito browser context
context = await browser.createIncognitoBrowserContext()
# Create a new page inside context
page = await context.newPage()
# ... do stuff with page ...
await page.goto('https://example.com')
# Dispose context once it's no longer needed
await context.close()
```

**browser**

Return the browser this browser context belongs to.

*coroutine* **close**() → None [source]

Close the browser context.

All the targets that belongs to the browser context will be closed.

> Note:
> Only incognito browser context can be closed.

**isIncognite**() → bool [source]

[Deprecated] Miss spelled method.

Use **isIncognito()** method instead.

**isIncognito**() → bool [source]

Return whether BrowserContext is incognito.

The default browser context is the only non-incognito browser context.

> Note:
> The default browser context cannot be closed.

*coroutine* **newPage**() → pyppeteer.page.Page [source]

Create a new page in the browser context.

**targets**() → List[pyppeteer.target.Target] [source]

Return a list of all active targets inside the browser context.

# Page Class

*class* pyppeteer.page.**Page**(*client: pyppeteer.connection.CDPSession, target: Target, frameTree: Dict[KT, VT], ignoreHTTPSErrors: bool, screenshotTaskQueue: list = None*) [source]

Bases: **pyee.EventEmitter**

Page class.

This class provides methods to interact with a single tab of chrome. One **Browser** object might have multiple Page object.

The **Page** class emits various **Events** which can be handled by using `on` or `once` method, which is inherited from pyee's `EventEmitter` class.

**Events** = *namespace(Close='close', Console='console', DOMContentLoaded='domcontentloaded', Dialog='dialog', Error='error', FrameAttached='frameattached', FrameDetached='framedetached', FrameNavigated='framenavigated', Load='load', Metrics='metrics', PageError='pageerror', Request='request', RequestFailed='requestfailed', RequestFinished='requestfinished', Response='response', WorkerCreated='workercreated', WorkerDestroyed='workerdestroyed')*

Available events.

*coroutine* **J**(*selector: str*) → Optional[pyppeteer.element_handle.ElementHandle]

alias to **querySelector()**

*coroutine* **JJ**(*selector: str*) → List[pyppeteer.element_handle.ElementHandle]

alias to **querySelectorAll()**

*coroutine* **JJeval**(*selector: str, pageFunction: str, *args*) → Any

alias to **querySelectorAllEval()**

*coroutine* **Jeval**(*selector: str, pageFunction: str, *args*) → Any

alias to **querySelectorEval()**

*coroutine* **Jx**(*expression: str*) → List[pyppeteer.element_handle.ElementHandle]

alias to **xpath()**

*coroutine* **addScriptTag**(*options: Dict[KT, VT] = None, **kwargs*) → pyppeteer.element_handle.ElementHandle [source]

Add script tag to this page.

One of `url`, `path` or `content` option is necessary.

- `url` (string): URL of a script to add.
- `path` (string): Path to the local JavaScript file to add.
- `content` (string): JavaScript string to add.
- `type` (string): Script type. Use `module` in order to load a JavaScript ES6 module.

**Return ElementHandle:**
**ElementHandle** of added tag.

*coroutine* **addStyleTag**(*options: Dict[KT, VT] = None*, ***kwargs*) → pyppeteer.element_handle.ElementHandle [source]

Add style or link tag to this page.

One of `url`, `path` or `content` option is necessary.

- `url` (string): URL of the link tag to add.
- `path` (string): Path to the local CSS file to add.
- `content` (string): CSS string to add.

**Return ElementHandle:**
`ElementHandle` of added tag.

*coroutine* **authenticate**(*credentials: Dict[str, str]*) → Any [source]

Provide credentials for http authentication.

`credentials` should be `None` or dict which has `username` and `password` field.

*coroutine* **bringToFront**() → None [source]

Bring page to front (activate tab).

**browser**

Get the browser the page belongs to.

*coroutine* **click**(*selector: str*, *options: dict = None*, ***kwargs*) → None [source]

Click element which matches `selector`.

This method fetches an element with `selector`, scrolls it into view if needed, and then uses `mouse` to click in the center of the element. If there's no element matching `selector`, the method raises `PageError`.

Available options are:

- `button` (str): `left`, `right`, or `middle`, defaults to `left`.
- `clickCount` (int): defaults to 1.
- `delay` (int | float): Time to wait between `mousedown` and `mouseup` in milliseconds. defaults to 0.

> Note:
> If this method triggers a navigation event and there's a separate `waitForNavigation()`, you may end up with a race condition that yields unexpected results. The correct pattern for click and wait for navigation is the following:
>
> ```
> await asyncio.gather(
>     page.waitForNavigation(waitOptions),
>     page.click(selector, clickOptions),
> )
> ```

*coroutine* **close**(*options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Close this page.

Available options:

- `runBeforeUnload` (bool): Defaults to `False`. Whether to run the before unload page handlers.

By defaults, **`close()` does not** run beforeunload handlers.

> **Note:**
> If `runBeforeUnload` is passed as `True`, a `beforeunload` dialog might be summoned and should be handled manually via page's `dialog` event.

*coroutine* **`content`**() → str [source]

Get the full HTML contents of the page.

Returns HTML including the doctype.

*coroutine* **`cookies`**(**urls*) → dict [source]

Get cookies.

If no URLs are specified, this method returns cookies for the current page URL. If URLs are specified, only cookies for those URLs are returned.

Returned cookies are list of dictionaries which contain these fields:

- `name` (str)
- `value` (str)
- `url` (str)
- `domain` (str)
- `path` (str)
- `expires` (number): Unix time in seconds
- `httpOnly` (bool)
- `secure` (bool)
- `session` (bool)
- `sameSite` (str): `'Strict'` or `'Lax'`

**`coverage`**

Return **`Coverage`**.

*coroutine* **`deleteCookie`**(**cookies*) → None [source]

Delete cookie.

`cookies` should be dictionaries which contain these fields:

- `name` (str): **required**
- `url` (str)
- `domain` (str)
- `path` (str)
- `secure` (bool)

*coroutine* **`emulate`**(*options: dict = None*, ***kwargs*) → None [source]

Emulate given device metrics and user agent.

This method is a shortcut for calling two methods:

- **`setUserAgent()`**
- **`setViewport()`**

`options` is a dictionary containing these fields:

- `viewport` (dict)
  - `width` (int): page width in pixels.
  - `height` (int): page width in pixels.
  - `deviceScaleFactor` (float): Specify device scale factor (can be thought as dpr). Defaults to 1.
  - `isMobile` (bool): Whether the `meta viewport` tag is taken into account. Defaults to `False`.
  - `hasTouch` (bool): Specifies if viewport supports touch events. Defaults to `False`.
  - `isLandscape` (bool): Specifies if viewport is in landscape mode. Defaults to `False`.
- `userAgent` (str): user agent string.

*coroutine* **`emulateMedia`**(*mediaType: str = None*) → None [source]

Emulate css media type of the page.

**Parameters:** **mediaType** (*str*) – Changes the CSS media type of the page. The only allowed values are `'screen'`, `'print'`, and `None`. Passing `None` disables media emulation.

*coroutine* **`evaluate`**(*pageFunction: str*, **args*, *force_expr: bool = False*) → Any [source]

Execute js-function or js-expression on browser and get result.

**Parameters:**
- **pageFunction** (*str*) – String of js-function/expression to be executed on the browser.
- **force_expr** (*bool*) – If True, evaluate **`pageFunction`** as expression. If False (default), try to automatically detect function or expression.

note: `force_expr` option is a keyword only argument.

*coroutine* **`evaluateHandle`**(*pageFunction: str*, **args*) → pyppeteer.execution_context.JSHandle

Execute function on this page. [source]

Difference between **`evaluate()`** and **`evaluateHandle()`** is that `evaluateHandle` returns JSHandle object (not value).

**Parameters:** **pageFunction** (*str*) – JavaScript function to be executed.

*coroutine* **`evaluateOnNewDocument`**(*pageFunction: str*, **args*) → None [source]

Add a JavaScript function to the document.

This function would be invoked in one of the following scenarios:

- whenever the page is navigated
- whenever the child frame is attached or navigated. In this case, the function is invoked in the context of the newly attached frame.

*coroutine* **`exposeFunction`**(*name: str*, *pyppeteerFunction: Callable[[...], Any]*) → None [source]

Add python function to the browser's `window` object as `name`.

Registered function can be called from chrome process.

**Parameters:**
- **name** (*string*) – Name of the function on the window object.
- **pyppeteerFunction** (*Callable*) – Function which will be called on python process. This function should not be asynchronous function.

*coroutine* **`focus`**(*selector: str*) → None [source]

Focus the element which matches `selector`.

If no element matched the `selector`, raise `PageError`.

**`frames`**

Get all frames of this page.

*coroutine* **`goBack`**(*options: dict = None*, ***kwargs*) → Optional[pyppeteer.network_manager.Response] [source]

Navigate to the previous page in history.

Available options are same as **`goto()`** method.

If cannot go back, return `None`.

*coroutine* **`goForward`**(*options: dict = None*, ***kwargs*) → Optional[pyppeteer.network_manager.Response] [source]

Navigate to the next page in history.

Available options are same as **`goto()`** method.

If cannot go forward, return `None`.

*coroutine* **`goto`**(*url: str*, *options: dict = None*, ***kwargs*) → Optional[pyppeteer.network_manager.Response] [source]

Go to the `url`.

**Parameters:** **url** (*string*) – URL to navigate page to. The url should include scheme, e.g. `https://`.

Available options are:

- `timeout` (int): Maximum navigation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the **`setDefaultNavigationTimeout()`** method.
- `waitUntil` (str | List[str]): When to consider navigation succeeded, defaults to `load`. Given a list of event strings, navigation is considered to be successful after all events have been fired. Events can be either:
  - `load`: when `load` event is fired.
  - `domcontentloaded`: when the `DOMContentLoaded` event is fired.
  - `networkidle0`: when there are no more than 0 network connections for at least 500 ms.
  - `networkidle2`: when there are no more than 2 network connections for at least 500 ms.

The `Page.goto` will raise errors if:

- there's an SSL error (e.g. in case of self-signed certificates)
- target URL is invalid
- the `timeout` is exceeded during navigation
- then main resource failed to load

> Note:
> **`goto()`** either raise error or return a main resource response. The only exceptions are navigation to `about:blank` or navigation to the same URL with a different hash, which would succeed and return `None`.

> **Note:**
> Headless mode doesn't support navigation to a PDF document.

*coroutine* **hover**(*selector: str*) → None [source]

Mouse hover the element which matches `selector`.

If no element matched the `selector`, raise `PageError`.

*coroutine* **injectFile**(*filePath: str*) → str [source]

[Deprecated] Inject file to this page.

This method is deprecated. Use **addScriptTag()** instead.

**isClosed**() → bool [source]

Indicate that the page has been closed.

**keyboard**

Get **Keyboard** object.

**mainFrame**

Get main **Frame** of this page.

*coroutine* **metrics**() → Dict[str, Any] [source]

Get metrics.

Returns dictionary containing metrics as key/value pairs:

- `Timestamp` (number): The timestamp when the metrics sample was taken.
- `Documents` (int): Number of documents in the page.
- `Frames` (int): Number of frames in the page.
- `JSEventListeners` (int): Number of events in the page.
- `Nodes` (int): Number of DOM nodes in the page.
- `LayoutCount` (int): Total number of full partial page layout.
- `RecalcStyleCount` (int): Total number of page style recalculations.
- `LayoutDuration` (int): Combined duration of page duration.
- `RecalcStyleDuration` (int): Combined duration of all page style recalculations.
- `ScriptDuration` (int): Combined duration of JavaScript execution.
- `TaskDuration` (int): Combined duration of all tasks performed by the browser.
- `JSHeapUsedSize` (float): Used JavaScript heap size.
- `JSHeapTotalSize` (float): Total JavaScript heap size.

**mouse**

Get **Mouse** object.

*coroutine* **pdf**(*options: dict = None*, ***kwargs*) → bytes [source]

Generate a pdf of the page.

Options:

- `path` (str): The file path to save the PDF.
- `scale` (float): Scale of the webpage rendering, defaults to `1`.
- `displayHeaderFooter` (bool): Display header and footer. Defaults to `False`.

- `headerTemplate` (str): HTML template for the print header. Should be valid HTML markup with following classes.
  - `date`: formatted print date
  - `title`: document title
  - `url`: document location
  - `pageNumber`: current page number
  - `totalPages`: total pages in the document
- `footerTemplate` (str): HTML template for the print footer. Should use the same template as `headerTemplate`.
- `printBackground` (bool): Print background graphics. Defaults to `False`.
- `landscape` (bool): Paper orientation. Defaults to `False`.
- `pageRanges` (string): Paper ranges to print, e.g., '1-5,8,11-13'. Defaults to empty string, which means all pages.
- `format` (str): Paper format. If set, takes priority over `width` or `height`. Defaults to `Letter`.
- `width` (str): Paper width, accepts values labeled with units.
- `height` (str): Paper height, accepts values labeled with units.
- `margin` (dict): Paper margins, defaults to `None`.
  - `top` (str): Top margin, accepts values labeled with units.
  - `right` (str): Right margin, accepts values labeled with units.
  - `bottom` (str): Bottom margin, accepts values labeled with units.
  - `left` (str): Left margin, accepts values labeled with units.

**Returns:** Return generated PDF `bytes` object.

> **Note:**
> Generating a pdf is currently only supported in headless mode.

**`pdf()`** generates a pdf of the page with `print` css media. To generate a pdf with `screen` media, call `page.emulateMedia('screen')` before calling **`pdf()`**.

> **Note:**
> By default, **`pdf()`** generates a pdf with modified colors for printing. Use the `--webkit-print-color-adjust` property to force rendering of exact colors.

```
await page.emulateMedia('screen')
await page.pdf({'path': 'page.pdf'})
```

The `width`, `height`, and `margin` options accept values labeled with units. Unlabeled values are treated as pixels.

A few examples:

- `page.pdf({'width': 100})`: prints with width set to 100 pixels.
- `page.pdf({'width': '100px'})`: prints with width set to 100 pixels.
- `page.pdf({'width': '10cm'})`: prints with width set to 100 centimeters.

All available units are:

- `px`: pixel
- `in`: inch

- `cm`: centimeter
- `mm`: millimeter

The format options are:

- `Letter`: 8.5in x 11in
- `Legal`: 8.5in x 14in
- `Tabloid`: 11in x 17in
- `Ledger`: 17in x 11in
- `A0`: 33.1in x 46.8in
- `A1`: 23.4in x 33.1in
- `A2`: 16.5in x 23.4in
- `A3`: 11.7in x 16.5in
- `A4`: 8.27in x 11.7in
- `A5`: 5.83in x 8.27in
- `A6`: 4.13in x 5.83in

> **Note:**
> `headerTemplate` and `footerTemplate` markup have the following limitations:
>
> 1. Script tags inside templates are not evaluated.
> 2. Page styles are not visible inside templates.

*coroutine* **`plainText`**() → str [source]

[Deprecated] Get page content as plain text.

*coroutine* **`queryObjects`**(*prototypeHandle: pyppeteer.execution_context.JSHandle*) → pyppeteer.execution_context.JSHandle [source]

Iterate js heap and finds all the objects with the handle.

**Parameters:** **prototypeHandle** (*JSHandle*) – JSHandle of prototype object.

*coroutine* **`querySelector`**(*selector: str*) → Optional[pyppeteer.element_handle.ElementHandle]

Get an Element which matches `selector`. [source]

**Parameters:** **selector** (*str*) – A selector to search element.
**Return Optional[ElementHandle]:**
If element which matches the `selector` is found, return its **`ElementHandle`**. If not found, returns `None`.

*coroutine* **`querySelectorAll`**(*selector: str*) → List[pyppeteer.element_handle.ElementHandle]

Get all element which matches `selector` as a list. [source]

**Parameters:** **selector** (*str*) – A selector to search element.
**Return List[ElementHandle]:**
List of **`ElementHandle`** which matches the `selector`. If no element is matched to the `selector`, return empty list.

*coroutine* **`querySelectorAllEval`**(*selector: str*, *pageFunction: str*, **args*) → Any [source]

Execute function with all elements which matches `selector`.

**Parameters:**
- **selector** (*str*) – A selector to query page for.

- **pageFunction** (*str*) – String of JavaScript function to be evaluated on browser. This function takes Array of the matched elements as the first argument.
- **args** (*Any*) – Arguments to pass to `pageFunction`.

*coroutine* **querySelectorEval**(*selector: str*, *pageFunction: str*, **args*) → Any [source]

Execute function with an element which matches `selector`.

**Parameters:**
- **selector** (*str*) – A selector to query page for.
- **pageFunction** (*str*) – String of JavaScript function to be evaluated on browser. This function takes an element which matches the selector as a first argument.
- **args** (*Any*) – Arguments to pass to `pageFunction`.

This method raises error if no element matched the `selector`.

*coroutine* **reload**(*options: dict = None*, ***kwargs*) → Optional[pyppeteer.network_manager.Response] [source]

Reload this page.

Available options are same as **goto()** method.

*coroutine* **screenshot**(*options: dict = None*, ***kwargs*) → Union[bytes, str] [source]

Take a screen shot.

The following options are available:

- `path` (str): The file path to save the image to. The screenshot type will be inferred from the file extension.
- `type` (str): Specify screenshot type, can be either `jpeg` or `png`. Defaults to `png`.
- `quality` (int): The quality of the image, between 0-100. Not applicable to `png` image.
- `fullPage` (bool): When true, take a screenshot of the full scrollable page. Defaults to `False`.
- `clip` (dict): An object which specifies clipping region of the page. This option should have the following fields:
  - `x` (int): x-coordinate of top-left corner of clip area.
  - `y` (int): y-coordinate of top-left corner of clip area.
  - `width` (int): width of clipping area.
  - `height` (int): height of clipping area.
- `omitBackground` (bool): Hide default white background and allow capturing screenshot with transparency.
- `encoding` (str): The encoding of the image, can be either `'base64'` or `'binary'`. Defaults to `'binary'`.

*coroutine* **select**(*selector: str*, **values*) → List[str] [source]

Select options and return selected values.

If no element matched the `selector`, raise `ElementHandleError`.

*coroutine* **setBypassCSP**(*enabled: bool*) → None [source]

Toggles bypassing page's Content-Security-Policy.

> Note:

CSP bypassing happens at the moment of CSP initialization rather then evaluation. Usually this means that `page.setBypassCSP` should be called before navigating to the domain.

*coroutine* **setCacheEnabled**(*enabled: bool = True*) → None [source]

Enable/Disable cache for each request.

By default, caching is enabled.

*coroutine* **setContent**(*html: str*) → None [source]

Set content to this page.

**Parameters:** **html** (*str*) – HTML markup to assign to the page.

*coroutine* **setCookie**(**cookies*) → None [source]

Set cookies.

`cookies` should be dictionaries which contain these fields:

- `name` (str): **required**
- `value` (str): **required**
- `url` (str)
- `domain` (str)
- `path` (str)
- `expires` (number): Unix time in seconds
- `httpOnly` (bool)
- `secure` (bool)
- `sameSite` (str): `'Strict'` or `'Lax'`

**setDefaultNavigationTimeout**(*timeout: int*) → None [source]

Change the default maximum navigation timeout.

This method changes the default timeout of 30 seconds for the following methods:

- **goto()**
- **goBack()**
- **goForward()**
- **reload()**
- **waitForNavigation()**

**Parameters:** **timeout** (*int*) – Maximum navigation time in milliseconds. Pass `0` to disable timeout.

*coroutine* **setExtraHTTPHeaders**(*headers: Dict[str, str]*) → None [source]

Set extra HTTP headers.

The extra HTTP headers will be sent with every request the page initiates.

> Note:
> `page.setExtraHTTPHeaders` does not guarantee the order of headers in the outgoing requests.

**Parameters:** **headers** (*Dict*) – A dictionary containing additional http headers to be sent with every requests. All header values must be string.

*coroutine* **setJavaScriptEnabled**(*enabled: bool*) → None [source]

Set JavaScript enable/disable.

*coroutine* **setOfflineMode**(*enabled: bool*) → None [source]

Set offline mode enable/disable.

*coroutine* **setRequestInterception**(*value: bool*) → None [source]

Enable/disable request interception.

Activating request interception enables **Request** class's **abort()**, **continue_()**, and **response()** methods. This provides the capability to modify network requests that are made by a page.

*coroutine* **setUserAgent**(*userAgent: str*) → None [source]

Set user agent to use in this page.

**Parameters:** **userAgent** (*str*) – Specific user agent to use in this page

*coroutine* **setViewport**(*viewport: dict*) → None [source]

Set viewport.

Available options are:

- `width` (int): page width in pixel.
- `height` (int): page height in pixel.
- `deviceScaleFactor` (float): Default to 1.0.
- `isMobile` (bool): Default to `False`.
- `hasTouch` (bool): Default to `False`.
- `isLandscape` (bool): Default to `False`.

*coroutine* **tap**(*selector: str*) → None [source]

Tap the element which matches the `selector`.

**Parameters:** **selector** (*str*) – A selector to search element to touch.

**target**

Return a target this page created from.

*coroutine* **title**() → str [source]

Get page's title.

**touchscreen**

Get **Touchscreen** object.

**tracing**

Get tracing object.

*coroutine* **type**(*selector: str*, *text: str*, *options: dict = None*, ***kwargs*) → None [source]

Type `text` on the element which matches `selector`.

If no element matched the `selector`, raise `PageError`.

Details see **pyppeteer.input.Keyboard.type()**.

**url**

Get URL of this page.

**viewport**

Get viewport as a dictionary.

Fields of returned dictionary is same as **setViewport()**.

**waitFor**(*selectorOrFunctionOrTimeout: Union[str, int, float], options: dict = None, *args, **kwargs*) → Awaitable[T_co] [source]

Wait for function, timeout, or element which matches on page.

This method behaves differently with respect to the first argument:

- If `selectorOrFunctionOrTimeout` is number (int or float), then it is treated as a timeout in milliseconds and this returns future which will be done after the timeout.
- If `selectorOrFunctionOrTimeout` is a string of JavaScript function, this method is a shortcut to **waitForFunction()**.
- If `selectorOrFunctionOrTimeout` is a selector string or xpath string, this method is a shortcut to **waitForSelector()** or **waitForXPath()**. If the string starts with `//`, the string is treated as xpath.

Pyppeteer tries to automatically detect function or selector, but sometimes miss-detects. If not work as you expected, use **waitForFunction()** or **waitForSelector()** directly.

**Parameters:**
- **selectorOrFunctionOrTimeout** – A selector, xpath, or function string, or timeout (milliseconds).
- **args** (*Any*) – Arguments to pass the function.

**Returns:** Return awaitable object which resolves to a JSHandle of the success value.

Available options: see **waitForFunction()** or **waitForSelector()**

**waitForFunction**(*pageFunction: str, options: dict = None, *args, **kwargs*) → Awaitable[T_co]

Wait until the function completes and returns a truthy value. [source]

**Parameters:** **args** (*Any*) – Arguments to pass to `pageFunction`.

**Returns:** Return awaitable object which resolves when the `pageFunction` returns a truthy value. It resolves to a **JSHandle** of the truthy value.

This method accepts the following options:

- `polling` (str | number): An interval at which the `pageFunction` is executed, defaults to `raf`. If `polling` is a number, then it is treated as an interval in milliseconds at which the function would be executed. If `polling` is a string, then it can be one of the following values:
  - `raf`: to constantly execute `pageFunction` in `requestAnimationFrame` callback. This is the tightest polling mode which is suitable to observe styling changes.
  - `mutation`: to execute `pageFunction` on every DOM mutation.
- `timeout` (int | float): maximum time to wait for in milliseconds. Defaults to 30000 (30 seconds). Pass `0` to disable timeout.

*coroutine* **waitForNavigation**(*options: dict = None, **kwargs*) → Optional[pyppeteer.network_manager.Response] [source]

Wait for navigation.

Available options are same as **goto()** method.

This returns `Response` when the page navigates to a new URL or reloads. It is useful for when you run code which will indirectly cause the page to navigate. In case of navigation to a different anchor or navigation due to History API usage, the navigation will return `None`.

Consider this example:

```
navigationPromise = async.ensure_future(page.waitForNavigation())
await page.click('a.my-link')  # indirectly cause a navigation
await navigationPromise  # wait until navigation finishes
```

or,

```
await asyncio.wait([
    page.click('a.my-link'),
    page.waitForNavigation(),
])
```

> **Note:**
> Usage of the History API to change the URL is considered a navigation.

*coroutine* **waitForRequest**(*urlOrPredicate: Union[str, Callable[[pyppeteer.network_manager.Request], bool]], options: Dict[KT, VT] = None, **kwargs*) → pyppeteer.network_manager.Request [source]

Wait for request.

**Parameters:** **urlOrPredicate** – A URL or function to wait for.

This method accepts below options:

- `timeout` (int | float): Maximum wait time in milliseconds, defaults to 30 seconds, pass `0` to disable the timeout.

Example:

```
firstRequest = await page.waitForRequest('http://example.com/resource')
finalRequest = await page.waitForRequest(lambda req: req.url == 'http://example.com' and
return firstRequest.url
```

*coroutine* **waitForResponse**(*urlOrPredicate: Union[str, Callable[[pyppeteer.network_manager.Response], bool]], options: Dict[KT, VT] = None, **kwargs*) → pyppeteer.network_manager.Response [source]

Wait for response.

**Parameters:** **urlOrPredicate** – A URL or function to wait for.

This method accepts below options:

- `timeout` (int | float): Maximum wait time in milliseconds, defaults to 30 seconds, pass `0` to disable the timeout.

Example:

```
firstResponse = await page.waitForResponse('http://example.com/resource')
finalResponse = await page.waitForResponse(lambda res: res.url == 'http://example.com' an
return finalResponse.ok
```

**waitForSelector**(*selector: str*, *options: dict = None*, ***kwargs*) → Awaitable[T_co] [source]

Wait until element which matches `selector` appears on page.

Wait for the `selector` to appear in page. If at the moment of calling the method the `selector` already exists, the method will return immediately. If the selector doesn't appear after the `timeout` milliseconds of waiting, the function will raise error.

**Parameters:** **selector** (*str*) – A selector of an element to wait for.
**Returns:** Return awaitable object which resolves when element specified by selector string is added to DOM.

This method accepts the following options:

- `visible` (bool): Wait for element to be present in DOM and to be visible; i.e. to not have `display: none` or `visibility: hidden` CSS properties. Defaults to `False`.
- `hidden` (bool): Wait for element to not be found in the DOM or to be hidden, i.e. have `display: none` or `visibility: hidden` CSS properties. Defaults to `False`.
- `timeout` (int | float): Maximum time to wait for in milliseconds. Defaults to 30000 (30 seconds). Pass `0` to disable timeout.

**waitForXPath**(*xpath: str*, *options: dict = None*, ***kwargs*) → Awaitable[T_co] [source]

Wait until element which matches `xpath` appears on page.

Wait for the `xpath` to appear in page. If the moment of calling the method the `xpath` already exists, the method will return immediately. If the xpath doesn't appear after `timeout` milliseconds of waiting, the function will raise exception.

**Parameters:** **xpath** (*str*) – A [xpath] of an element to wait for.
**Returns:** Return awaitable object which resolves when element specified by xpath string is added to DOM.

Available options are:

- `visible` (bool): wait for element to be present in DOM and to be visible, i.e. to not have `display: none` or `visibility: hidden` CSS properties. Defaults to `False`.
- `hidden` (bool): wait for element to not be found in the DOM or to be hidden, i.e. have `display: none` or `visibility: hidden` CSS properties. Defaults to `False`.
- `timeout` (int | float): maximum time to wait for in milliseconds. Defaults to 30000 (30 seconds). Pass `0` to disable timeout.

**workers**

Get all workers of this page.

*coroutine* **xpath**(*expression: str*) → List[pyppeteer.element_handle.ElementHandle] [source]

Evaluate the XPath expression.

If there are no such elements in this page, return an empty list.

**Parameters:** **expression** (*str*) – XPath string to be evaluated.

# Worker Class

*class* `pyppeteer.worker.`**Worker**(*client: CDPSession*, *url: str*, *consoleAPICalled: Callable[[str, List[pyppeteer.execution_context.JSHandle]], None]*, *exceptionThrown: Callable[[Dict[KT, VT]], None]*) [source]

Bases: **pyee.EventEmitter**

The Worker class represents a WebWorker.

The events **workercreated** and **workerdestroyed** are emitted on the page object to signal the worker lifecycle.

```
page.on('workercreated', lambda worker: print('Worker created:', worker.url))
```

*coroutine* **evaluate**(*pageFunction: str*, **args*) → Any [source]

Evaluate `pageFunction` with `args`.

Shortcut for `(await worker.executionContext).evaluate(pageFunction, *args)`.

*coroutine* **evaluateHandle**(*pageFunction: str*, **args*) → pyppeteer.execution_context.JSHandle

Evaluate `pageFunction` with `args` and return **JSHandle**. [source]

Shortcut for `(await worker.executionContext).evaluateHandle(pageFunction, *args)`.

*coroutine* **executionContext**() → pyppeteer.execution_context.ExecutionContext [source]

Return ExecutionContext.

**url**

Return URL.

# Keyboard Class

*class* `pyppeteer.input.`**Keyboard**(*client: pyppeteer.connection.CDPSession*) [source]

Bases: **object**

Keyboard class provides as api for managing a virtual keyboard.

The high level api is **type()**, which takes raw characters and generate proper keydown, keypress/input, and keyup events on your page.

For finer control, you can use **down()**, **up()**, and **sendCharacter()** to manually fire events as if they were generated from a real keyboard.

An example of holding down `Shift` in order to select and delete some text:

```
await page.keyboard.type('Hello, World!')
await page.keyboard.press('ArrowLeft')

await page.keyboard.down('Shift')
for i in ' World':
    await page.keyboard.press('ArrowLeft')
await page.keyboard.up('Shift')

await page.keyboard.press('Backspace')
# Result text will end up saying 'Hello!'.
```

An example of pressing `A`:

```
await page.keyboard.down('Shift')
await page.keyboard.press('KeyA')
await page.keyboard.up('Shift')
```

*coroutine* **down**(*key: str*, *options: dict = None*, ***kwargs*) → None [source]

Dispatch a `keydown` event with `key`.

If `key` is a single character and no modifier keys besides `Shift` are being held down, and a `keypress`/`input` event will also generated. The `text` option can be specified to force an `input` event to be generated.

If `key` is a modifier key, like `Shift`, `Meta`, or `Alt`, subsequent key presses will be sent with that modifier active. To release the modifier key, use **up()** method.

**Parameters:**
- **key** (*str*) – Name of key to press, such as `ArrowLeft`.
- **options** (*dict*) – Option can have `text` field, and if this option specified, generate an input event with this text.

> **Note:**
> Modifier keys DO influence **down()**. Holding down `shift` will type the text in upper case.

*coroutine* **press**(*key: str*, *options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Press `key`.

If `key` is a single character and no modifier keys besides `Shift` are being held down, a `keypress`/`input` event will also generated. The `text` option can be specified to force an input event to be generated.

**Parameters:** **key** (*str*) – Name of key to press, such as `ArrowLeft`.

This method accepts the following options:

- `text` (str): If specified, generates an input event with this text.
- `delay` (int | float): Time to wait between `keydown` and `keyup`. Defaults to 0.

> **Note:**
> Modifier keys DO effect **press()**. Holding down `Shift` will type the text in upper case.

*coroutine* **sendCharacter**(*char: str*) → None [source]

Send character into the page.

This method dispatches a `keypress` and `input` event. This does not send a `keydown` or `keyup` event.

> **Note:**
> Modifier keys DO NOT effect **sendCharacter()**. Holding down `shift` will not type the text in upper case.

*coroutine* **type**(*text: str*, *options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Type characters into a focused element.

This method sends `keydown`, `keypress/input`, and `keyup` event for each character in the `text`.

To press a special key, like `Control` or `ArrowDown`, use **`press()`** method.

**Parameters:**
- **text** (*str*) – Text to type into a focused element.
- **options** (*dict*) – Options can have `delay` (int | float) field, which specifies time to wait between key presses in milliseconds. Defaults to 0.

> **Note:**
> Modifier keys DO NOT effect **`type()`**. Holding down `shift` will not type the text in upper case.

*coroutine* **`up`**(*key: str*) → None [source]

Dispatch a `keyup` event of the `key`.

**Parameters:** **key** (*str*) – Name of key to release, such as `ArrowLeft`.

# Mouse Class

*class* `pyppeteer.input.`**`Mouse`**(*client: pyppeteer.connection.CDPSession, keyboard: pyppeteer.input.Keyboard*) [source]

Bases: **`object`**

Mouse class.

*coroutine* **`click`**(*x: float, y: float, options: dict = None, **kwargs*) → None [source]

Click button at (`x`, `y`).

Shortcut to **`move()`**, **`down()`**, and **`up()`**.

This method accepts the following options:

- `button` (str): `left`, `right`, or `middle`, defaults to `left`.
- `clickCount` (int): defaults to 1.
- `delay` (int | float): Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.

*coroutine* **`down`**(*options: dict = None, **kwargs*) → None [source]

Press down button (dispatches `mousedown` event).

This method accepts the following options:

- `button` (str): `left`, `right`, or `middle`, defaults to `left`.
- `clickCount` (int): defaults to 1.

*coroutine* **`move`**(*x: float, y: float, options: dict = None, **kwargs*) → None [source]

Move mouse cursor (dispatches a `mousemove` event).

Options can accepts `steps` (int) field. If this `steps` option specified, Sends intermediate `mousemove` events. Defaults to 1.

*coroutine* **`up`**(*options: dict = None, **kwargs*) → None [source]

Release pressed button (dispatches `mouseup` event).

This method accepts the following options:

- `button` (str): `left`, `right`, or `middle`, defaults to `left`.
- `clickCount` (int): defaults to 1.

# Tracing Class

*class* `pyppeteer.tracing.`**`Tracing`**(*client: pyppeteer.connection.CDPSession*) [source]

Bases: **`object`**

Tracing class.

You can use **`start()`** and **`stop()`** to create a trace file which can be opened in Chrome DevTools or timeline viewer.

```
await page.tracing.start({'path': 'trace.json'})
await page.goto('https://www.google.com')
await page.tracing.stop()
```

*coroutine* **`start`**(*options: dict = None*, ***kwargs*) → None [source]

Start tracing.

Only one trace can be active at a time per browser.

This method accepts the following options:

- `path` (str): A path to write the trace file to.
- `screenshots` (bool): Capture screenshots in the trace.
- `categories` (List[str]): Specify custom categories to use instead of default.

*coroutine* **`stop`**() → str [source]

Stop tracing.

**Returns:** trace data as string.

# Dialog Class

*class* `pyppeteer.dialog.`**`Dialog`**(*client: pyppeteer.connection.CDPSession*, *type: str*, *message: str*, *defaultValue: str = ''*) [source]

Bases: **`object`**

Dialog class.

Dialog objects are dispatched by page via the `dialog` event.

An example of using `Dialog` class:

```
browser = await launch()
page = await browser.newPage()

async def close_dialog(dialog):
    print(dialog.message)
    await dialog.dismiss()
    await browser.close()
```

```
page.on(
    'dialog',
    lambda dialog: asyncio.ensure_future(close_dialog(dialog))
)
await page.evaluate('() => alert("1")')
```

*coroutine* **accept**(*promptText: str = ''*) → None [source]

Accept the dialog.

- `promptText` (str): A text to enter in prompt. If the dialog's type is not prompt, this does not cause any effect.

**defaultValue**

If dialog is prompt, get default prompt value.

If dialog is not prompt, return empty string (`''`).

*coroutine* **dismiss**() → None [source]

Dismiss the dialog.

**message**

Get dialog message.

**type**

Get dialog type.

One of `alert`, `beforeunload`, `confirm`, or `prompt`.

# ConsoleMessage Class

*class* `pyppeteer.page.`**ConsoleMessage**(*type: str*, *text: str*, *args: List[pyppeteer.execution_context.JSHandle] = None*) [source]

Bases: **object**

Console message class.

ConsoleMessage objects are dispatched by page via the `console` event.

**args**

Return list of args (JSHandle) of this message.

**text**

Return text representation of this message.

**type**

Return type of this message.

# Frame Class

*class* `pyppeteer.frame_manager.`**Frame**(*client: pyppeteer.connection.CDPSession*, *parentFrame: Optional[Frame]*, *frameId: str*) [source]

Bases: **object**

Frame class.

Frame objects can be obtained via **pyppeteer.page.Page.mainFrame**.

*coroutine* **J**(*selector: str*) → Optional[pyppeteer.element_handle.ElementHandle]
Alias to **querySelector()**

*coroutine* **JJ**(*selector: str*) → List[pyppeteer.element_handle.ElementHandle]
Alias to **querySelectorAll()**

*coroutine* **JJeval**(*selector: str*, *pageFunction: str*, **args*) → Optional[Dict[KT, VT]]
Alias to **querySelectorAllEval()**

*coroutine* **Jeval**(*selector: str*, *pageFunction: str*, **args*) → Any
Alias to **querySelectorEval()**

*coroutine* **Jx**(*expression: str*) → List[pyppeteer.element_handle.ElementHandle]
Alias to **xpath()**

*coroutine* **addScriptTag**(*options: Dict[KT, VT]*) → pyppeteer.element_handle.ElementHandle
Add script tag to this frame. [source]

Details see **pyppeteer.page.Page.addScriptTag()**.

*coroutine* **addStyleTag**(*options: Dict[KT, VT]*) → pyppeteer.element_handle.ElementHandle
Add style tag to this frame. [source]

Details see **pyppeteer.page.Page.addStyleTag()**.

**childFrames**
Get child frames.

*coroutine* **click**(*selector: str*, *options: dict = None*, ***kwargs*) → None [source]
Click element which matches `selector`.

Details see **pyppeteer.page.Page.click()**.

*coroutine* **content**() → str [source]
Get the whole HTML contents of the page.

*coroutine* **evaluate**(*pageFunction: str*, **args*, *force_expr: bool = False*) → Any [source]
Evaluate pageFunction on this frame.

Details see **pyppeteer.page.Page.evaluate()**.

*coroutine* **evaluateHandle**(*pageFunction: str*, **args*) → pyppeteer.execution_context.JSHandle
Execute function on this frame. [source]

Details see **pyppeteer.page.Page.evaluateHandle()**.

*coroutine* **executionContext**() → Optional[pyppeteer.execution_context.ExecutionContext]
Return execution context of this frame. [source]

Return **ExecutionContext** associated to this frame.

*coroutine* **focus**(*selector: str*) → None [source]

Focus element which matches `selector`.

Details see **pyppeteer.page.Page.focus()**.

*coroutine* **hover**(*selector: str*) → None [source]

Mouse hover the element which matches `selector`.

Details see **pyppeteer.page.Page.hover()**.

*coroutine* **injectFile**(*filePath: str*) → str [source]

[Deprecated] Inject file to the frame.

**isDetached**() → bool [source]

Return `True` if this frame is detached.

Otherwise return `False`.

**name**

Get frame name.

**parentFrame**

Get parent frame.

If this frame is main frame or detached frame, return `None`.

*coroutine* **querySelector**(*selector: str*) → Optional[pyppeteer.element_handle.ElementHandle]

Get element which matches **selector** string. [source]

Details see **pyppeteer.page.Page.querySelector()**.

*coroutine* **querySelectorAll**(*selector: str*) → List[pyppeteer.element_handle.ElementHandle]

Get all elements which matches **selector**. [source]

Details see **pyppeteer.page.Page.querySelectorAll()**.

*coroutine* **querySelectorAllEval**(*selector: str*, *pageFunction: str*, **args*) → Optional[Dict[KT, VT]]

Execute function on all elements which matches selector. [source]

Details see **pyppeteer.page.Page.querySelectorAllEval()**.

*coroutine* **querySelectorEval**(*selector: str*, *pageFunction: str*, **args*) → Any [source]

Execute function on element which matches selector.

Details see **pyppeteer.page.Page.querySelectorEval()**.

*coroutine* **select**(*selector: str*, **values*) → List[str] [source]

Select options and return selected values.

Details see **pyppeteer.page.Page.select()**.

*coroutine* **setContent**(*html: str*) → None [source]

Set content to this page.

*coroutine* **tap**(*selector: str*) → None [source]

Tap the element which matches the `selector`.

Details see **`pyppeteer.page.Page.tap()`**.

*coroutine* **`title`**() → str [source]

Get title of the frame.

*coroutine* **`type`**(*selector: str*, *text: str*, *options: dict = None*, ***kwargs*) → None [source]

Type `text` on the element which matches `selector`.

Details see **`pyppeteer.page.Page.type()`**.

**`url`**

Get url of the frame.

**`waitFor`**(*selectorOrFunctionOrTimeout: Union[str, int, float]*, *options: dict = None*, **args*, ***kwargs*) → Union[Awaitable[T_co], pyppeteer.frame_manager.WaitTask] [source]

Wait until **`selectorOrFunctionOrTimeout`**.

Details see **`pyppeteer.page.Page.waitFor()`**.

**`waitForFunction`**(*pageFunction: str*, *options: dict = None*, **args*, ***kwargs*) → pyppeteer.frame_manager.WaitTask [source]

Wait until the function completes.

Details see **`pyppeteer.page.Page.waitForFunction()`**.

**`waitForSelector`**(*selector: str*, *options: dict = None*, ***kwargs*) → pyppeteer.frame_manager.WaitTask [source]

Wait until element which matches `selector` appears on page.

Details see **`pyppeteer.page.Page.waitForSelector()`**.

**`waitForXPath`**(*xpath: str*, *options: dict = None*, ***kwargs*) → pyppeteer.frame_manager.WaitTask [source]

Wait until element which matches `xpath` appears on page.

Details see **`pyppeteer.page.Page.waitForXPath()`**.

*coroutine* **`xpath`**(*expression: str*) → List[pyppeteer.element_handle.ElementHandle] [source]

Evaluate the XPath expression.

If there are no such elements in this frame, return an empty list.

**Parameters:** **expression** (*str*) – XPath string to be evaluated.

# ExecutionContext Class

*class* `pyppeteer.execution_context.`**`ExecutionContext`**(*client: pyppeteer.connection.CDPSession*, *contextPayload: Dict[KT, VT]*, *objectHandleFactory: Any*, *frame: Frame = None*) [source]

Bases: **`object`**

Execution Context class.

*coroutine* **evaluate**(*pageFunction: str*, **args*, *force_expr: bool = False*) → Any [source]

Execute `pageFunction` on this context.

Details see **pyppeteer.page.Page.evaluate()**.

*coroutine* **evaluateHandle**(*pageFunction: str*, **args*, *force_expr: bool = False*) → pyppeteer.execution_context.JSHandle [source]

Execute `pageFunction` on this context.

Details see **pyppeteer.page.Page.evaluateHandle()**.

**frame**

Return frame associated with this execution context.

*coroutine* **queryObjects**(*prototypeHandle: pyppeteer.execution_context.JSHandle*) → pyppeteer.execution_context.JSHandle [source]

Send query.

Details see **pyppeteer.page.Page.queryObjects()**.

# JSHandle Class

*class* `pyppeteer.execution_context.`**JSHandle**(*context: pyppeteer.execution_context.ExecutionContext*, *client: pyppeteer.connection.CDPSession*, *remoteObject: Dict[KT, VT]*) [source]

Bases: **object**

JSHandle class.

JSHandle represents an in-page JavaScript object. JSHandle can be created with the **evaluateHandle()** method.

**asElement**() → Optional[ElementHandle] [source]

Return either null or the object handle itself.

*coroutine* **dispose**() → None [source]

Stop referencing the handle.

**executionContext**

Get execution context of this handle.

*coroutine* **getProperties**() → Dict[str, pyppeteer.execution_context.JSHandle] [source]

Get all properties of this handle.

*coroutine* **getProperty**(*propertyName: str*) → pyppeteer.execution_context.JSHandle [source]

Get property value of `propertyName`.

*coroutine* **jsonValue**() → Dict[KT, VT] [source]

Get Jsonized value of this object.

**toString**() → str [source]

Get string representation.

# ElementHandle Class

*class* `pyppeteer.element_handle.`**`ElementHandle`**(*context: pyppeteer.execution_context.ExecutionContext, client: pyppeteer.connection.CDPSession, remoteObject: dict, page: Any, frameManager: FrameManager*) [source]

Bases: **`pyppeteer.execution_context.JSHandle`**

ElementHandle class.

This class represents an in-page DOM element. ElementHandle can be created by the **`pyppeteer.page.Page.querySelector()`** method.

ElementHandle prevents DOM element from garbage collection unless the handle is disposed. ElementHandles are automatically disposed when their origin frame gets navigated.

ElementHandle isinstance can be used as arguments in **`pyppeteer.page.Page.querySelectorEval()`** and **`pyppeteer.page.Page.evaluate()`** methods.

*coroutine* **`J`**(*selector: str*) → Optional[pyppeteer.element_handle.ElementHandle]

alias to **`querySelector()`**

*coroutine* **`JJ`**(*selector: str*) → List[pyppeteer.element_handle.ElementHandle]

alias to **`querySelectorAll()`**

*coroutine* **`JJeval`**(*selector: str, pageFunction: str, *args*) → Any

alias to **`querySelectorAllEval()`**

*coroutine* **`Jeval`**(*selector: str, pageFunction: str, *args*) → Any

alias to **`querySelectorEval()`**

*coroutine* **`Jx`**(*expression: str*) → List[pyppeteer.element_handle.ElementHandle]

alias to **`xpath()`**

**`asElement`**() → pyppeteer.element_handle.ElementHandle [source]

Return this ElementHandle.

*coroutine* **`boundingBox`**() → Optional[Dict[str, float]] [source]

Return bounding box of this element.

If the element is not visible, return `None`.

This method returns dictionary of bounding box, which contains:

- `x` (int): The X coordinate of the element in pixels.
- `y` (int): The Y coordinate of the element in pixels.
- `width` (int): The width of the element in pixels.
- `height` (int): The height of the element in pixels.

*coroutine* **`boxModel`**() → Optional[Dict[KT, VT]] [source]

Return boxes of element.

Return `None` if element is not visible. Boxes are represented as an list of points; each Point is a dictionary `{x, y}`. Box points are sorted clock-wise.

Returned value is a dictionary with the following fields:

- `content` (List[Dict]): Content box.
- `padding` (List[Dict]): Padding box.
- `border` (List[Dict]): Border box.
- `margin` (List[Dict]): Margin box.
- `width` (int): Element's width.
- `height` (int): Element's height.

*coroutine* **`click`**(*options: dict = None*, ***kwargs*) → None [source]

Click the center of this element.

If needed, this method scrolls element into view. If the element is detached from DOM, the method raises `ElementHandleError`.

`options` can contain the following fields:

- `button` (str): `left`, `right`, of `middle`, defaults to `left`.
- `clickCount` (int): Defaults to 1.
- `delay` (int | float): Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.

*coroutine* **`contentFrame`**() → Optional[pyppeteer.frame_manager.Frame] [source]

Return the content frame for the element handle.

Return `None` if this handle is not referencing iframe.

*coroutine* **`focus`**() → None [source]

Focus on this element.

*coroutine* **`hover`**() → None [source]

Move mouse over to center of this element.

If needed, this method scrolls element into view. If this element is detached from DOM tree, the method raises an `ElementHandleError`.

*coroutine* **`isIntersectingViewport`**() → bool [source]

Return `True` if the element is visible in the viewport.

*coroutine* **`press`**(*key: str*, *options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Press `key` onto the element.

This method focuses the element, and then uses **`pyppeteer.input.keyboard.down()`** and **`pyppeteer.input.keyboard.up()`**.

**Parameters:** **key** (*str*) – Name of key to press, such as `ArrowLeft`.

This method accepts the following options:

- `text` (str): If specified, generates an input event with this text.
- `delay` (int | float): Time to wait between `keydown` and `keyup`. Defaults to 0.

*coroutine* **`querySelector`**(*selector: str*) → Optional[pyppeteer.element_handle.ElementHandle]

Return first element which matches `selector` under this element. [source]

If no element matches the `selector`, returns `None`.

*coroutine* **querySelectorAll**(*selector: str*) → List[pyppeteer.element_handle.ElementHandle] [source]

Return all elements which match `selector` under this element.

If no element matches the `selector`, returns empty list (`[]`).

*coroutine* **querySelectorAllEval**(*selector: str*, *pageFunction: str*, **args*) → Any [source]

Run `Page.querySelectorAllEval` within the element.

This method runs `Array.from(document.querySelectorAll)` within the element and passes it as the first argument to `pageFunction`. If there is no element matching `selector`, the method raises `ElementHandleError`.

If `pageFunction` returns a promise, then wait for the promise to resolve and return its value.

Example:

```
<div class="feed">
    <div class="tweet">Hello!</div>
    <div class="tweet">Hi!</div>
</div>
```

```
feedHandle = await page.J('.feed')
assert (await feedHandle.JJeval('.tweet', '(nodes => nodes.map(n => n.innerText))')) == [
```

*coroutine* **querySelectorEval**(*selector: str*, *pageFunction: str*, **args*) → Any [source]

Run `Page.querySelectorEval` within the element.

This method runs `document.querySelector` within the element and passes it as the first argument to `pageFunction`. If there is no element matching `selector`, the method raises `ElementHandleError`.

If `pageFunction` returns a promise, then wait for the promise to resolve and return its value.

`ElementHandle.Jeval` is a shortcut of this method.

Example:

```
tweetHandle = await page.querySelector('.tweet')
assert (await tweetHandle.querySelectorEval('.like', 'node => node.innerText')) == 100
assert (await tweetHandle.Jeval('.retweets', 'node => node.innerText')) == 10
```

*coroutine* **screenshot**(*options: Dict[KT, VT] = None*, ***kwargs*) → bytes [source]

Take a screenshot of this element.

If the element is detached from DOM, this method raises an `ElementHandleError`.

Available options are same as **`pyppeteer.page.Page.screenshot()`**.

*coroutine* **tap**() → None [source]

Tap the center of this element.

If needed, this method scrolls element into view. If the element is detached from DOM, the method raises `ElementHandleError`.

*coroutine* **type**(*text: str*, *options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Focus the element and then type text.

Details see **pyppeteer.input.Keyboard.type()** method.

*coroutine* **uploadFile**(**filePaths*) → dict [source]

Upload files.

*coroutine* **xpath**(*expression: str*) → List[pyppeteer.element_handle.ElementHandle] [source]

Evaluate the XPath expression relative to this elementHandle.

If there are no such elements, return an empty list.

**Parameters:** **expression** (*str*) – XPath string to be evaluated.

# Request Class

*class* pyppeteer.network_manager.**Request**(*client: pyppeteer.connection.CDPSession, requestId: Optional[str], interceptionId: str, isNavigationRequest: bool, allowInterception: bool, url: str, resourceType: str, payload: dict, frame: Optional[pyppeteer.frame_manager.Frame], redirectChain: List[Request]*) [source]

Bases: **object**

Request class.

Whenever the page sends a request, such as for a network resource, the following events are emitted by pyppeteer's page:

- `'request'`: emitted when the request is issued by the page.
- `'response'`: emitted when/if the response is received for the request.
- `'requestfinished'`: emitted when the response body is downloaded and the request is complete.

If request fails at some point, then instead of `'requestfinished'` event (and possibly instead of `'response'` event), the `'requestfailed'` event is emitted.

If request gets a `'redirect'` response, the request is successfully finished with the `'requestfinished'` event, and a new request is issued to a redirect url.

*coroutine* **abort**(*errorCode: str = 'failed'*) → None [source]

Abort request.

To use this, request interception should be enabled by **pyppeteer.page.Page.setRequestInterception()**. If request interception is not enabled, raise `NetworkError`.

`errorCode` is an optional error code string. Defaults to `failed`, could be one of the following:

- `aborted`: An operation was aborted (due to user action).
- `accessdenied`: Permission to access a resource, other than the network, was denied.
- `addressunreachable`: The IP address is unreachable. This usually means that there is no route to the specified host or network.
- `blockedbyclient`: The client chose to block the request.
- `blockedbyresponse`: The request failed because the request was delivered along with requirements which are not met ('X-Frame-Options' and 'Content-Security-Policy'

ancestor check, for instance).

- `connectionaborted`: A connection timeout as a result of not receiving an ACK for data sent.
- `connectionclosed`: A connection was closed (corresponding to a TCP FIN).
- `connectionfailed`: A connection attempt failed.
- `connectionrefused`: A connection attempt was refused.
- `connectionreset`: A connection was reset (corresponding to a TCP RST).
- `internetdisconnected`: The Internet connection has been lost.
- `namenotresolved`: The host name could not be resolved.
- `timedout`: An operation timed out.
- `failed`: A generic failure occurred.

*coroutine* **`continue_`**(*overrides: Dict[KT, VT] = None*) → None [source]

Continue request with optional request overrides.

To use this method, request interception should be enabled by **`pyppeteer.page.Page.setRequestInterception()`**. If request interception is not enabled, raise `NetworkError`.

`overrides` can have the following fields:

- `url` (str): If set, the request url will be changed.
- `method` (str): If set, change the request method (e.g. `GET`).
- `postData` (str): If set, change the post data or request.
- `headers` (dict): If set, change the request HTTP header.

**`failure`**() → Optional[Dict[KT, VT]] [source]

Return error text.

Return `None` unless this request was failed, as reported by `requestfailed` event.

When request failed, this method return dictionary which has a `errorText` field, which contains human-readable error message, e.g. `'net::ERR_RAILED'`.

**`frame`**

Return a matching **`frame`** object.

Return `None` if navigating to error page.

**`headers`**

Return a dictionary of HTTP headers of this request.

All header names are lower-case.

**`isNavigationRequest`**() → bool [source]

Whether this request is driving frame's navigation.

**`method`**

Return this request's method (GET, POST, etc.).

**`postData`**

Return post body of this request.

**`redirectChain`**

Return chain of requests initiated to fetch a resource.

- If there are no redirects and request was successful, the chain will be empty.
- If a server responds with at least a single redirect, then the chain will contain all the requests that were redirected.

`redirectChain` is shared between all the requests of the same chain.

**`resourceType`**

Resource type of this request perceived by the rendering engine.

ResourceType will be one of the following: `document`, `stylesheet`, `image`, `media`, `font`, `script`, `texttrack`, `xhr`, `fetch`, `eventsource`, `websocket`, `manifest`, `other`.

*coroutine* **`respond`**(*response: Dict[KT, VT]*) → None [source]

Fulfills request with given response.

To use this, request interception should by enabled by **`pyppeteer.page.Page.setRequestInterception()`**. Request interception is not enabled, raise `NetworkError`.

`response` is a dictionary which can have the following fields:

- `status` (int): Response status code, defaults to 200.
- `headers` (dict): Optional response headers.
- `contentType` (str): If set, equals to setting `Content-Type` response header.
- `body` (str | bytes): Optional response body.

**`response`**

Return matching **`Response`** object, or `None`.

If the response has not been received, return `None`.

**`url`**

URL of this request.

# Response Class

*class* `pyppeteer.network_manager.`**`Response`**(*client: pyppeteer.connection.CDPSession, request: pyppeteer.network_manager.Request, status: int, headers: Dict[str, str], fromDiskCache: bool, fromServiceWorker: bool, securityDetails: Dict[KT, VT] = None*) [source]

Bases: **`object`**

Response class represents responses which are received by `Page`.

**`buffer`**() → Awaitable[bytes] [source]

Return awaitable which resolves to bytes with response body.

**`fromCache`**

Return `True` if the response was served from cache.

Here **`cache`** is either the browser's disk cache or memory cache.

**`fromServiceWorker`**

Return `True` if the response was served by a service worker.

**headers**

Return dictionary of HTTP headers of this response.

All header names are lower-case.

*coroutine* **json**() → dict [source]

Get JSON representation of response body.

**ok**

Return bool whether this request is successful (200-299) or not.

**request**

Get matching **Request** object.

**securityDetails**

Return security details associated with this response.

Security details if the response was received over the secure connection, or **None** otherwise.

**status**

Status code of the response.

*coroutine* **text**() → str [source]

Get text representation of response body.

**url**

URL of the response.

# Target Class

*class* pyppeteer.browser.**Target**(*targetInfo: Dict[KT, VT], browserContext: BrowserContext, sessionFactory: Callable[[], Coroutine[Any, Any, pyppeteer.connection.CDPSession]], ignoreHTTPSErrors: bool, setDefaultViewport: bool, screenshotTaskQueue: List[T], loop: asyncio.events.AbstractEventLoop*) [source]

Bases: **object**

Browser's target class.

**browser**

Get the browser the target belongs to.

**browserContext**

Return the browser context the target belongs to.

*coroutine* **createCDPSession**() → pyppeteer.connection.CDPSession [source]

Create a Chrome Devtools Protocol session attached to the target.

**opener**

Get the target that opened this target.

Top-level targets return None.

*coroutine* **page**() → Optional[pyppeteer.page.Page] [source]

Get page of this target.

If the target is not of type “page” or “background_page”, return `None`.

**type**

Get type of this target.

Type can be `'page'`, `'background_page'`, `'service_worker'`, `'browser'`, or `'other'`.

**url**

Get url of this target.

# CDPSession Class

*class* `pyppeteer.connection.`**`CDPSession`**(*connection: Union[pyppeteer.connection.Connection, CDPSession], targetType: str, sessionId: str, loop: asyncio.events.AbstractEventLoop*) [source]

Bases: **`pyee.EventEmitter`**

Chrome Devtools Protocol Session.

The **`CDPSession`** instances are used to talk raw Chrome Devtools Protocol:

- protocol methods can be called with **`send()`** method.
- protocol events can be subscribed to with **`on()`** method.

Documentation on DevTools Protocol can be found here.

*coroutine* **`detach`**() → None [source]

Detach session from target.

Once detached, session won’t emit any events and can’t be used to send messages.

**`send`**(*method: str*, *params: dict = None*) → Awaitable[T_co] [source]

Send message to the connected session.

**Parameters:**
- **method** (*str*) – Protocol method name.
- **params** (*dict*) – Optional method parameters.

# Coverage Class

*class* `pyppeteer.coverage.`**`Coverage`**(*client: pyppeteer.connection.CDPSession*) [source]

Bases: **`object`**

Coverage class.

Coverage gathers information about parts of JavaScript and CSS that were used by the page.

An example of using JavaScript and CSS coverage to get percentage of initially executed code:

```
# Enable both JavaScript and CSS coverage
await page.coverage.startJSCoverage()
await page.coverage.startCSSCoverage()

# Navigate to page
await page.goto('https://example.com')
```

```
# Disable JS and CSS coverage and get results
jsCoverage = await page.coverage.stopJSCoverage()
cssCoverage = await page.coverage.stopCSSCoverage()
totalBytes = 0
usedBytes = 0
coverage = jsCoverage + cssCoverage
for entry in coverage:
    totalBytes += len(entry['text'])
    for range in entry['ranges']:
        usedBytes += range['end'] - range['start'] - 1

print('Bytes used: {}%'.format(usedBytes / totalBytes * 100))
```

*coroutine* **startCSSCoverage**(*options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Start CSS coverage measurement.

Available options are:

- `resetOnNavigation` (bool): Whether to reset coverage on every navigation. Defaults to `True`.

*coroutine* **startJSCoverage**(*options: Dict[KT, VT] = None*, ***kwargs*) → None [source]

Start JS coverage measurement.

Available options are:

- `resetOnNavigation` (bool): Whether to reset coverage on every navigation. Defaults to `True`.
- `reportAnonymousScript` (bool): Whether anonymous script generated by the page should be reported. Defaults to `False`.

> **Note:**
> Anonymous scripts are ones that don't have an associated url. These are scripts that are dynamically created on the page using `eval` of `new Function`. If `reportAnonymousScript` is set to `True`, anonymous scripts will have `__pyppeteer_evaluation_script__` as their url.

*coroutine* **stopCSSCoverage**() → List[T] [source]

Stop CSS coverage measurement and get result.

Return list of coverage reports for all non-anonymous scripts. Each report includes:

- `url` (str): StyleSheet url.
- `text` (str): StyleSheet content.
- `ranges` (List[Dict]): StyleSheet ranges that were executed. Ranges are sorted and non-overlapping.
  - `start` (int): A start offset in text, inclusive.
  - `end` (int): An end offset in text, exclusive.

> **Note:**
> CSS coverage doesn't include dynamically injected style tags without sourceURLs (but currently includes... to be fixed).

*coroutine* **stopJSCoverage**() → List[T] [source]

Stop JS coverage measurement and get result.

Return list of coverage reports for all scripts. Each report includes:

- `url` (str): Script url.
- `text` (str): Script content.
- `ranges` (List[Dict]): Script ranges that were executed. Ranges are sorted and non-overlapping.
  - `start` (int): A start offset in text, inclusive.
  - `end` (int): An end offset in text, exclusive.

> **Note:**
> JavaScript coverage doesn't include anonymous scripts by default. However, scripts with sourceURLs are reported.

# Debugging

For debugging, you can set `logLevel` option to `logging.DEBUG` for **`pyppeteer.launcher.launch()`** and **`pyppeteer.launcher.connect()`** functions. However, this option prints too many logs including SEND/RECV messages of pyppeteer. In order to only show suppressed error messages, you should set `pyppeteer.DEBUG` to `True`.

Example:

```
import asyncio
import pyppeteer
from pyppeteer import launch

pyppeteer.DEBUG = True  # print suppressed errors as error log

async def main():
    browser = await launch()
    ...  # do something

asyncio.get_event_loop().run_until_complete(main())
```