DisCapTy

Release 2.0

Predeactor

Feb 21, 2023

CONTENTS

DisCaPty is a Python module to generate Captcha images & challenges without struggling your mind on how to make your own.

Features:

- Highly customizable Captcha generation / Hackable
- Developer & user friendly
- Extendable with your own/third-party generators

CHAPTER ONE

INSTALLATION

To install DisCapTy, run the following command in your wished environment:

```
pip install -U DisCapTy
```

With Poetry:

```
peotry add DisCapTy
```

CHAPTER

TWO

EXAMPLE

```
from discapty import Challenge, ImageGenerator, TooManyRetriesError

challenge = Challenge(ImageGenerator(width=500, height=250, fonts=["./my-fonts.ttf"]),␣
↪allowed_retries=5)
captcha = challenge.begin()
send_captcha_to_user(captcha)

user_input = get_user_input()

try:
    is_valid = challenge.check(user_input)
    if is_valid:
        print("You're correct!")
    else:
        print("You're incorrect, try again!")
        # And here, do something to get user's input & check again.
except TooManyRetriesError:
    print("You've made too many errors!")
```

CHAPTER

THREE

PAGES

3.1 Introduction

This is a simplified guide to show everyone how to use the DisCapTy library. This guide is meant for everyone, however if you're an experienced developer, you might be interested to read the *DisCapTy's API* directly.

3.1.1 To read this guide

Before you read this guide, we need to explain some terms we will use here. The word we will define here will be reference in uppercase and italic through this guide, for example, the word *CAPTCHA*, defined below, will be used in the guide as "*CAPTCHA*".

1. `GENERATOR`: A *GENERATOR* is a class that can be initialized with required/optional arguments (Or none at all), and that will generate a *CAPTCHA OBJECT* based on any given input. The result can be of any form.
2. `RAW CODE/CODE`: The *RAW CODE* or *CODE* is the code that you'll provide to DisCapTy, it is a clear code, and it is the expected input of your user.
3. `CAPTCHA OBJECT`: The *CAPTCHA OBJECT* is the result of a *GENERATOR*, it can be anything (Text, audio, image, etc. . .) and it is what the user will face during it's *CHALLENGE*.
4. `CHALLENGE`: A *CHALLENGE* is a temporary question-answer made for one single user. A *CHALLENGE* can have different states:
   - `PENDING`: The *CHALLENGE* is waiting to begin.
   - `WAITING`: The *CHALLENGE* has begun, he is now waiting for user's input.
   - `COMPLETED`: The *CHALLENGE* has been completed with success.
   - `FAILED`: The *CHALLENGE* has been failed by the user. (Generally by giving too many wrong answers)
   - `FAILURE`: The *CHALLENGE* has unexpectedly failed (Such as cancelled)

   It is generally thrown away after being completed, or failed.
5. `CAPTCHA CLASS`: Refers to the `discapty.Captcha` class.
6. `CHALLENGE CLASS`: Refers to the `discapty.Challenge` class.

3.1.2 Objects of DisCapTy

Assuming you're reading this guide because this is the first time you're interacting with DisCapTy, you may need to understand what objects will DisCapTy serves you. This is important because if your codebase doesn't understand what you are using, you might find yourself into a mess that is not a captcha.

`discapty.Captcha`

A *CAPTCHA CLASS* contain the *RAW CODE* and it's *CAPTCHA OBJECT* in the same place. It is where you can check for user's input directly using the `.check` function.

> **Attention:** This class does not generates the *CAPTCHA OBJECT* itself, a *GENERATOR* do. The *CAPTCHA CLASS* just wrap the *RAW CODE* and the *CAPTCHA OBJECT* together.

`discapty.Challenge`

The *CHALLENGE CLASS* is the DisCapTy's implementation of a *CHALLENGE*. With the *CHALLENGE CLASS*, you are able to generate the captcha, verify's user input, set a defined limit of retries, use a custom *CODE*, etc…

The *CHALLENGE CLASS* has different states, as stated in *CHALLENGE*. If the challenge state's is either `FAILED`, `FAILURE` or `COMPLETED`, it cannot be edited. While properties are writable, you're advised to not touch them manually.

You can access the states in *`discapty.States`*

Subclasses of `discapty.Generator`

Any subclasses of `discapty.Generator` are considered to be *GENERATORS*. They can be used in *`Challenges`*, or directly, like this:

```python
# TextGenerator is a subclass of Generator
from discapty import TextGenerator

captcha_object = TextGenerator().generate('My Code')

send_to_user(captcha_object)
```

A generator can have default arguments arguments. You can change them directly when initializing the class:

```python
# An image based Captcha object generator
from discapty import WheezyGenerator

captcha_object = WheezyGenerator(width=500, height=200, noise_level=3).generate('My code
↪')  # Returns a PIL.Image.Image object

send_image_to_user(captcha_object)
```

Certain generators will requires you to give certain arguments. In the case of DisCapTy's builtin generators, they all have optional arguments.

3.1.3 Creating a Challenge

Now that you know what you'll interact with, it's time for you to create your first *CHALLENGE CLASS*.

To create a *CHALLENGE*, you just have to initialize the *CHALLENGE CLASS* with an initialized generator you want to use.

```
from discapty import Challenge, TextGenerator

challenge = Challenge(TextGenerator())
captcha_object = challenge.begin()  # You'll obtain your CAPTCHA_OBJECT HERE
```

From here you can send your *CAPTCHA OBJECT* to your user, and you can validate the user's input like this:

```
user_input = get_user_input()

is_valid_input = challenge.check(user_input)
```

This is a basic example, and it is a *bad* one, because the `.check` function can raise *TooManyRetriesError* if `.check` has been used more than the `allowed_retries` attributes allows it. The `allowed_retries` attribute can be edited when creating the *CHALLENGE CLASS*.

If you do like a more complete example, check the following:

```
from discapty import Challenge, TextGenerator, TooManyRetriesError

challenge = Challenge(TextGenerator(), allowed_retries=3)

first_captcha = challenge.begin()
send_to_user(first_captcha)

# challenge.is_completed returns `True` when the Challenge's state is either completed or␣
↪failed.
while not challenge.is_completed:
    user_input = get_user_input()

    try:
        is_right = challenge.check(user_input)
        # If it is right, the challenge will be completed.
    except TooManyRetriesError:
        # From here, challenge will be completed.
        is_right = False

    # The loop will continue until a right answer has been completed or if there is too␣
↪many retries.

if is_right:
    do_something_for_completing_the_captcha()
else:
    do_something_for_failing_the_captcha()
```

This code is already more suitable for your needs.

3.1.4 Creating a Captcha queue

The DisCapTy's Captcha queue permit the developers to store many *CHALLENGE CLASS* in one place, it takes cares of managing all of them. Putting in place the Captcha queue is fairly easy. The Captcha queue will always give an ID to a challenge, if you don't pass one, an UUID will be generated for you.

To use the queue, as always you just need to initialize it with one or more initialized generator(s):

```python
from discapty import CaptchaQueue, WheezyGenerator, TextGenerator

# With one generator
my_queue = CaptchaQueue(TextGenerator())

# With multiple generators
my_queue = CaptchaQueue([TextGenerator(), WheezyGenerator()])
```

if you use multiple generators, this mean that one generator will be picked randomly when creating a *CHALLENGE CLASS*.

Warning: This may create inconsistency when generating *CAPTCHA OBJECTS* where you'll need to check in your code what kind of *CAPTCHA OBJECT* you receive, for example, you may send an image differently from a string.

After then, you can create a challenge by calling `.create_challenge`:

```python
from discapty import CaptchaQueue, TextGenerator

queue = CaptchaQueue(TextGenerator())

challenge = queue.create_challenge()  # You'll obtain a challenge here

send_captcha_to_user(challenge.captcha)

challenge_id = challenge.id
# To obtain your challenge through it's ID
challenge = queue.get_challenge(challenge_id)

# To delete/cancel your challenge
queue.delete_challenge(challenge_id)
```

3.2 Builtin generators

Here is the following generators coming with DisCapTy:

3.2.1 TextGenerator

`class discapty.TextGenerator`(*, *separator: Union[str, List[str]] = '\u200b'*)

A text-based Captcha generator. Most insecure, but it is the most tricky.

It adds a specific separator between each character of the given text. The default separator is an invisible space. (\u200B)

Important: The following generators are image-based generators, meaning you'll receive images. If you use the color arguments, we make use of pydantic.Color. While you CAN pass a `str` object, you IDE might complain that you didn't pass a `pydantic.Color` object. This is fine, you can just ignore this error, your string will be processed without trouble.

3.2.2 WheezyGenerator

`class discapty.WheezyGenerator`(*, *fonts: Sequence[Union[pydantic.types.FilePath, str]] = ['/home/docs/checkouts/readthedocs.org/user_builds/discapty/envs/2.0/lib/python3.10/site-packages/discapty/fonts/Roboto-Regular.ttf'], fonts_size: Tuple[int, ...] = (50,), width: int = 300, height: int = 125, background_color: pydantic.color.Color = '#EEEECC', text_color: pydantic.color.Color = '#5C87B2', text_squeeze_factor: float = 0.8, noise_number: int = 30, noise_color: pydantic.color.Color = '#EEEECC', noise_level: int = 2*)

A wheezy image Captcha generator. Comes with many customizable settings. Easier to read than Image.

Example: https://imgur.com/a/l9V09PN

3.2.3 ImageGenerator

`class discapty.ImageGenerator`(*, *fonts: Sequence[Union[pydantic.types.FilePath, str]] = ['/home/docs/checkouts/readthedocs.org/user_builds/discapty/envs/2.0/lib/python3.10/site-packages/discapty/fonts/Roboto-Regular.ttf'], fonts_size: Tuple[int, ...] = (50,), background_color: pydantic.color.Color = '#F8FEFA00', text_color: pydantic.color.Color = '#8A8C1A64', number_of_dots: int = 100, width_of_dots: int = 3, number_of_curves: int = 1, width: int = 300, height: int = 125*)

An image Captcha generator. Comes with many customizable settings. More harder than the Wheezy generator.

Example: https://imgur.com/a/wozYgW0

3.3 DisCapTy's API

3.3.1 Generator

class discapty.generators.Generator

Base class for all generators.

A generator is used to especially generate a Captcha object based on a given text. A generator looks like this:

```python
class MyGenerator(Generator):
    def generate(self, text: str) -> str:
        return "+".join(text)
```

A generator can be supplied with parameters using class's attributes, for example:

```python
class MyGenerator(Generator):
    separator = "+"

    def generate(self, text: str) -> str:
        return self.separator.join(text)

gen1 = MyGenerator()  # Separator here is "+"
gen2 = MyGenerator(separator="-")  # Separator here is "-"
```

Here, separator has a default value, which can be overridden by the user, or not. If you wish to create a generator with a required value, you can use "...", like this:

```python
class MyGenerator(Generator):
    separator: str = ...


    ...

MyGenerator(separator="+")  # Works!
MyGenerator()  # Raises an error!
```

If you wish to know more on that subject, visit Pydantic's documentation as this is what `Generator` uses under the hood. https://pydantic-docs.helpmanual.io/

New in version 2.0.0.

property required_keys: List[str]

Returns a list of all child's required keys.

property optional_keys: List[str]

Returns a list of all child's optional keys.

abstract generate(*text: str*) → Any

A method that needs to be implemented by the child class. This method will return the Captcha that the user has requested. See class's docstring.

3.3.2 Captcha

`class discapty.`**`Captcha`**`(`*code: str*, *captcha_object: Any*`)`

Represent a Captcha object.

`property code: str`

The code in clear of the Captcha.

`property captcha: Any`

The captcha object. This is what's send to the user.

`property type: Any`

The type of the captcha object. It is the same as doing `type(self.captcha)`.

Changed in version 2.0.0: The Captcha object is no longer what creates the Captcha image, it just is the representation of the Captcha that the user will face.

`check``(`*text: str*, ***, *force_casing: bool = False*, *remove_spaces: bool = True*`)` → bool

Check if a text is correct to the captcha code.

Parameters

- **`text`** (*`str`*) – The answer to check against the Captcha's code.
- **`force_casing`** (*`bool`*) – If True, the casing must be respected. Defaults to False.
- **`remove_spaces`** (*`bool`*) – If True, spaces will be removed when checking the answer. Defaults to True.

Returns True if the answer is correct, False otherwise.

Return type bool

3.3.3 Challenge

`class discapty.`**`Challenge`**`(`*generator:* discapty.generators.Generator, *challenge_id: Optional[str] = None*, ***, *allowed_retries: Optional[int] = None*, *code: Optional[str] = None*, *code_length: Optional[int] = None*`)`

Representation of a challenge. A challenge represent the user's Captcha question-answer he must face.

This class takes cares of:

- Generating the captcha
- Verify inputs
- Manage the "Captcha" object

It frees your mind from managing all the process of a captcha challenge, keeping your code short and easy.

`property code: str`

The raw code of the Challenge.

`property allowed_retries: int`

The number of allowed retries.

`property failures: int`

The number of failures the user has realized.

property attempted_tries: int

The number of tries the user has attempted. (Or how many time was `.begin` called)

property state: *discapty.States*

The current state of the challenge.

property fail_reason: str

The reason why the challenge has failed. Only filled if the state is failed or failure.

Parameters

- **generator** (Generator) – The generator class to use. You cannot uses *discapty.generators.Generator* directly, you have to subclass it and implement the "generate" function first.
- **challenge_id** (*Optional[int | str]*) – The id of the challenge. Can be a string or an id. If none is supplied, a random UUID will be generated.
- **allowed_retries** (*Optional[int]*) – The number of retries allowed. Defaults to 3.
- **code** (*Optional[str]*) – The code to use. If none is supplied, a random code will be generated.
- **code_length** (*int*) – The length of the code to generate if no code is supplied. Defaults to 4.

New in version 2.0.0.

property captcha_object: Any

Get the Captcha object.

Returns The Captcha object.

Return type `typing.Any`

property captcha: *discapty.captcha.Captcha*

Returns the Captcha class associated to this challenge.

property is_completed: bool

Check if the challenge has been completed or failed.

property is_correct: Optional[bool]

Check if the challenge has been completed. If not, return None. If failed, return False.

property is_wrong: Optional[bool]

Check if the challenge has been failed. If not, return None. If completed, return False.

begin() → Any

Start the challenge.

Returns The Captcha object to send to the user.

Return type *Captcha*

Raises

- ***AlreadyCompletedError*** – If the challenge has already been completed. You cannot start a challenge twice, you need to create a new one.
- ***AlreadyRunningError*** – If the challenge is already running.

- *TooManyRetriesError* – If the number of failures is greater than the number of retries allowed. In other words, the challenge has failed.
- *ChallengeCompletionError* – If the challenge had a failure. Returns the failure's reason.

check(*answer: str*, *, *force_casing: bool = False*, *remove_spaces: bool = True*) → bool

Check an answer. This will always add +1 to *attempted_tries* and *failures* if necessary.

Parameters

- **answer** (*str*) – The answer to check against the Captcha's code.
- **force_casing** (*bool*) – If True, the casing must be respected. Defaults to False.
- **remove_spaces** (*bool*) – If True, spaces will be removed when checking the answer. Defaults to True.

Returns True if the answer is correct, False otherwise.

Return type bool

Raises

- *TooManyRetriesError* – If the number of failures is greater than the number of retries allowed. We are still adding +1 to the failure even when raising the exception.
- **TypeError** – The challenge cannot be edited (State is either not PENDING or not WAITING)

reload(*, *increase_attempted_tries: bool = True*, *increase_failures: bool = False*) → Any

Reload the Challenge and its code.

This method will create a new random code. It will also increase the attempted_tries counter if requested. By defaults, this behavior is executed.

Parameters

- **increase_attempted_tries** (*bool*) – If True, the attempted_tries counter will be increased.
- **increase_failures** (*bool*) – If True, the failures counter will be increased.

Raises **TypeError** – If the challenge cannot be edited or is not already running.

cancel() → None

Cancel the challenge.

Raises **TypeError:** – If the challenge cannot be edited.

3.3.4 CaptchaQueue

class discapty.**CaptchaQueue**(*generators: Union[discapty.generators.Generator, List[discapty.generators.Generator]]*, *, *queue: Optional[Dict[str, discapty.challenge.Challenge]] = None*)

A safe handler for taking cares of managing the challenges for the developer.

It basically offers a sane & internal way to manage your captcha using a key-value pair without ever having to touch the challenges/captcha directly.

Parameters

- **generators** (Generator | *List*[Generator]) – A list or a single generator to use for creating the challenges. If a list is given, a random generator will be picked up when using *create_challenge*.

  You should be aware that inconsistency will occur this way, as if one generator can return a specific type and another one could return another kind of type.

- **queue** (*Dict*[*str*, Challenge]) – Import an existing queue. Shouldn't be required.

Raises **ValueError** – If no generators has been passed.

create_challenge(*challenge_id: Optional[str] = None*, ***, *retries: Optional[int] = None*, *code: Optional[str] = None*, *code_length: Optional[int] = None*) → *discapty.challenge.Challenge*

Create a challenge for an id. Overwrite the challenge created before, unless the challenge is not fully completed.

Parameters

- **challenge_id** (*str*) – The id associated to the challenge. If not given, a random id will be generated.
- **retries** (*int*) – The number of allowed retries. Defaults to 3.
- **code** (*str*) – The code to use. Defaults to a random code.
- **code_length** (*int*) – The length of the code to generate if no code is supplied. Defaults to 4.

Returns The generated challenge.

Return type *Challenge*

get_challenge(*challenge_id: str*) → *discapty.challenge.Challenge*

Get the challenge of an id, if it exist.

Parameters **challenge_id** (*str*) – The id associated to the challenge.

Returns The challenge associated to the id.

Return type *Challenge*

Raises **UnexistingChallengeError** – If the given id does not have any associated challenge.

delete_challenge(*challenge_id: str*) → None

Delete a challenge of an id, if it exist.

Parameters **challenge_id** (*int*) – The id associated to the challenge.

Raises **UnexistingChallengeError** – If the given id does not have any associated challenge.

3.3.5 States

States are only used with `discapty.Challenge`.

class discapty.**States**(*value*)

An enum representing the different states of a challenge.

Available states are:

- PENDING : The challenge is waiting to begin.
- WAITING : The challenge is waiting for user's input.

- COMPLETED : The challenge has been completed.
- FAILED : The challenge has been failed without trouble.
- FAILURE : The challenge has been completed without user's input and in an unexpected way. (e.g. manually cancelled)

3.3.6 Errors

exception `discapty.errors.`**`NonexistingChallengeError`**

Raised when trying to get a challenge that does not exist. Subclass of "KeyError" as this error will appear when trying to get the challenge from a dict.

exception `discapty.errors.`**`InvalidFontError`**

Raised when one or more fonts are invalid.

exception `discapty.errors.`**`ChallengeCompletionError`**

Raised when a challenge has an issue regarding its completion.

exception `discapty.errors.`**`TooManyRetriesError`**

Raised when a challenge received more retries than allowed.

exception `discapty.errors.`**`AlreadyCompletedError`**

Raised when a challenge has already been completed.

exception `discapty.errors.`**`AlreadyRunningError`**

Raised when a challenge is already running.

3.4 Migration Guide

3.4.1 From 1.0.x to 2.0

DisCapTy has been created to be a supplementary tools for `discord.py`, however, its owner had announced that this library would not be supported anymore. From here, many peoples has created forks of this library, which was making DisCapTy completely unusable for others library. This thought has dragged me to think about what DisCapTy should be & become. As such, it has been decided that DisCapTy should NOT be related to `discord.py` anymore but as a standalone library with no restriction on where it could be used.

The most important points are:

1. Deprecating generating Captcha in the *`discapty.Captcha`* class, but rather in a `discapty.Generator` subclass.
2. Featuring *`discapty.Challenge`* & *`discapty.CaptchaQueue`*.
3. Added more specific errors to the library.
4. A documentation has been created.

Rewrite of Captcha class

The *discapty.Captcha* object does not longer generates the Captcha object anymore, what does is a generator. There is no exact alternative to generates the Captcha object in the same place as the Captcha class, since now the Captcha class only **include** the Captcha object and its code. However, you can do this:

Regarding diff block

The "-" represent the old version, the "+" represent the actual, new version.

```
--- /home/docs/checkouts/readthedocs.org/user_builds/discapty/checkouts/2.0/docs/source/
↪docs/source/code_sample/captcha_object/captcha_object_20.old.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/discapty/checkouts/2.0/docs/source/
↪docs/source/code_sample/captcha_object/captcha_object_20.py
@@ -1,7 +1,10 @@
 import discapty

-captcha = discapty.Captcha("whezzy")
-captcha_object = captcha.generate_captcha()
+code = "My code"
+generator = discapty.WheezyGenerator()
+
+captcha_object = generator.generate(code)
+captcha = discapty.Captcha(code, captcha_object)

 # Checking the code
-is_correct = captcha.verify_code(user_input)
+is_correct: bool = captcha.check(user_input)
```

Removal of `.setup` function

Along with the rewrite of the Captcha class, the `.setup` function has been removed, instead, parameters can be provided to a generator when initializing a generator class.

```
--- /home/docs/checkouts/readthedocs.org/user_builds/discapty/checkouts/2.0/docs/source/
↪docs/source/code_sample/generator_init/gen_init_20.old.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/discapty/checkouts/2.0/docs/source/
↪docs/source/code_sample/generator_init/gen_init_20.py
@@ -1,4 +1,4 @@
 import discapty

-captcha = discapty.Captcha("wheezy")
-captcha.setup(width=200, height=100)
+generator = discapty.WheezyGenerator(width=200, height=100)
+# Do the rest...
```

Added `Challenge` class

The *discapty.Challenge* class is the new preferred way to create Captcha now. You can read more about challenges here: *Introduction to Challenges* - *Creating a Challenge*

To use `Challenge` rather than the old `Captcha`, you can do these changes:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/discapty/checkouts/2.0/docs/source/
↪docs/source/code_sample/captcha_to_challenge/c_to_c.old.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/discapty/checkouts/2.0/docs/source/
↪docs/source/code_sample/captcha_to_challenge/c_to_c.py
@@ -1,9 +1,8 @@
-import discapty
+from discapty import Challenge, WheezyGenerator

-captcha = discapty.Captcha("whezzy")
-captcha.setup(width=200, height=100)
+challenge = Challenge(WheezyGenerator(width=200, height=100))

-captcha_object = captcha.generate_captcha()
+captcha_object = challenge.begin()

 # Checking the code
-is_correct = captcha.verify_code(user_input)
+is_correct = challenge.check(user_input)
```

PYTHON MODULE INDEX

INDEX