土台編
（後編）

PythonやJupyterでiPhone/iPad先端機能を簡単・自由にプログラミング！

Carnet for Jupyter, Juno, with Pythonista and Pyto

－ iOSフレームワークをPython/Jupyterで動的/自由に使う －

平林　純

土台篇（後編）について

iPhone/iPad デバイスが備えるさまざまな機能、それらを「Pythonから使えるようにしたい！」と思った時に、それを実現するための「前提知識や方法、あるいは、試行錯誤のエッセンス」をまとめたものが、本書「土台編（後編）」です。本書「土台編（後編）」の主人公は、意味や価値を生み出す演者のために、影で働く裏方の人たちです。

本書が「土台」すなわち、ライブラリ（パッケージ・モジュール）を作るなら、実際に使って色々な応用を試すのは本書の前編にあたる「活用編」です。こちらの方は、どちらかというと、意味や価値を生み出す演者に近いかもしれません。

どんな道具も「使ってこそ意味や価値がある」ものです。つまりは、「活用編（前編）」「土台編（後編）」まとめて、楽しんで頂ければ幸いです。

平林　純　2022/08/31

目次

目次

1. iPhoneやiPadを動かす「仕組み」を知っておく

1. iPhoneやiPadを動かす「仕組み」を知っておく

本書の前編にあたる「活用編」では、事前に作りおきした「iOSデバイス機能をPythonから使うための3分クッキング的パッケージ」を使うことで、手軽に簡単にiOSプログラミングを楽しみました。iPhoneやiPadのさまざまなハードウェア・ソフトウェア機能を、Pythonで書かれた便利なパッケージやモジュールを使うことで、色々な活用例を手軽に実現してみました。

後編にあたる本書は、題して「土台篇」です。「iOSのソフト・ハード機能をPythonから使うことができる」ようにするための、Pythonプログラミング情報を詰め込んでみました。つまり、前編で使ったパッケージやモジュールの実装例を題材にしつつ、PythonからiPhoneやiPad機能にアクセスするための方法・テクニックをまとめました。

既存パッケージやモジュールをカスタマイズした場合でも、あるいはApple社が提供する新機能をPythonから使えるようにする場合でも、どちらの場合に対しても、本書内容は役立つはずです。

1.1 iOSやiPadOS を分類すると UNIX系のOSに属してる

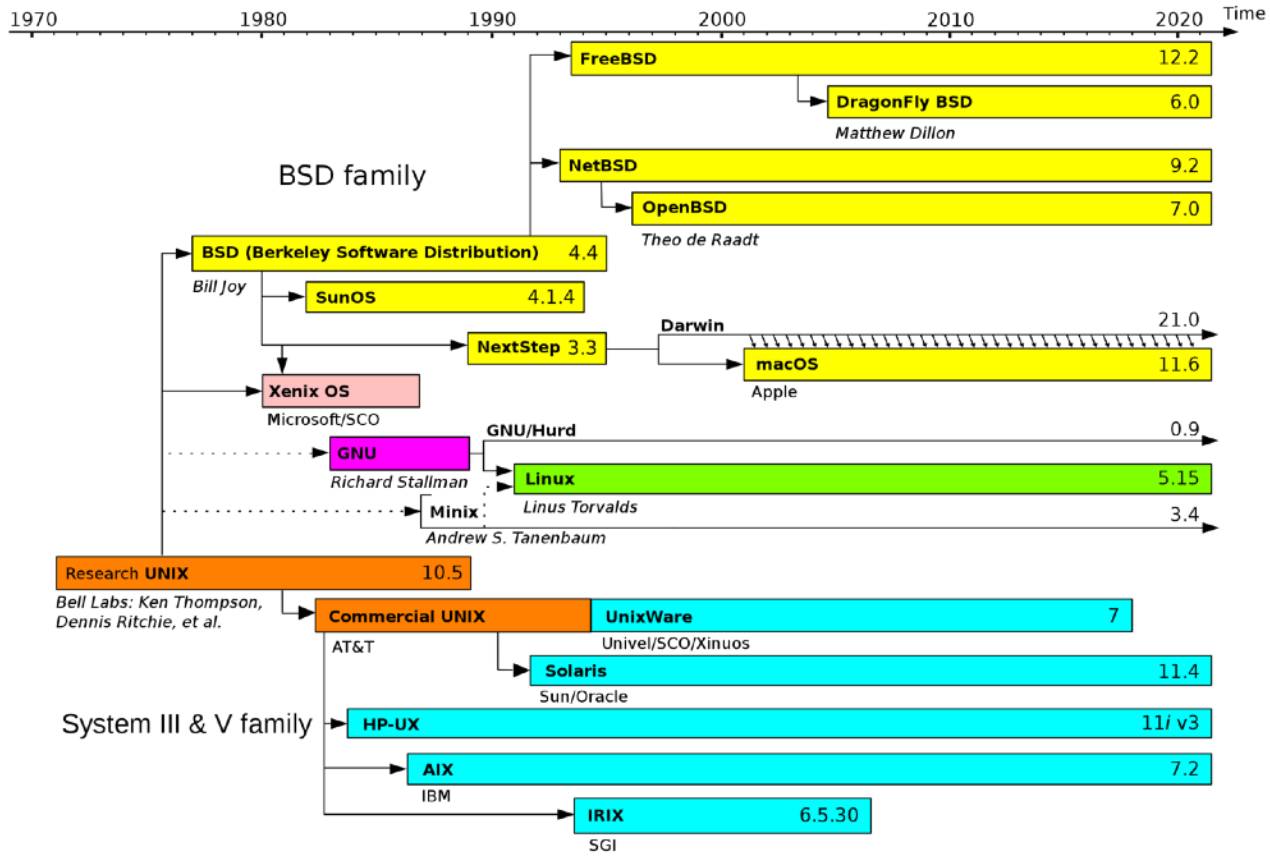
iPhoneやiPadの機能をPythonから自由に使うためには、iPhoneやiPadのことをよく知っていなければなりません。その理由は、Python世界とiPhone・iPad世界の間で「上手い通訳」をするためには、両方の世界に対する理解が必要となるから、です。Pythonのことだけでなく、iPhoneやiPadのこともわかっていなければなりません。というわけで、まずは『iPhoneやiPadを動かす「仕組み」を知っておく』ことにしましょう。

iPhoneやiPadを動かしているiOSやiPadOSは、Unix系のOSに属しています。iOS・iPad OS・macOS…それらはNeXTコンピュータ社が開発したBSD UNIX系OSであるNeXTSTEPの流れを汲んでいます。

iOS・iPad OS・macOSのもっとも基礎になる部分はDarwin カーネル（OS）とも呼ばれています。Darwinは、カーネギーメロン大学で開発されたMachカーネルやFreeBSDなどを元に作られたもので、基本機能部分（カーネル）のソースコードは（すべてのコードではないものの）公開もされています*。

* https://github.com/apple/darwin-xnu

各種OSの系譜



(https://en.wikipedia.org/wiki/Darwin_(operating_system))

iOSやiPadOSがUnix系OSであることを確認するために、Python環境アプリから次のPythonコードを実行してみましょう。これは、「動いているカーネルの名前や情報を返す unameコマンド」を動かして、その結果を出力するコードです。

ノートブック：_carnets_system.ipynb

```
import os

result = os.popen('uname -a'); print(result.read())
```

```
Darwin iPad 21.6.0 Darwin Kernel Version 21.6.0: Sat Jun 18 18:56:4
8 PDT 2022; root:xnu-8020.140.41~4/RELEASE_ARM64_T8010 iPad7,6
```

たとえば手元のiPadの場合には、上記のような出力がされます。出力結果を眺めれば、「”Darwin Kernel Version 21.6.0”が動いている」ということがわかります。

このように、iOSやiPad OS のベースでは、UNIX系のOS（OS基礎部分＝カーネル）のDarwinが動いているわけです。

さて、iPhoneやiPadのベースがUNIX系OSというのなら、Linuxなどでよく使うコマンドも使えるかもしれません。そこで、

・カレントディレクトリを表示するpwdコマンド
・ディレクトリ内にあるファイルを一覧表示するlsコマンド

をPythonから次のように実行させてみます。

```
result = os.popen('pwd'); print(result.read())
result = os.popen('ls'); print(result.read())
```

```
/private/var/mobile/Library/Mobile Documents/com~apple~CloudDocs/Py
thon

DepthPrediction.jpg
DepthPrediction.png
Mona_Lisa.jpg
_arnets_manualCapture_old_but work.ipynb
```

すると、たとえば上記のように、カレントディレクトリの絶対パスやファイル一覧が出力されてます。iPhoneやiPadは、（デフォルト状態では機能制限も多いのですが）UNIX系のOSが動く、由緒正しいコンピューターというわけです。

1.2 macOSと同じくiOS/iPadOSでも使える音声発生コマンド

iOSとiPadOSはmacOSを元にして作られています。そのため、macOSで使うことができるコマンドライン機能*も（すべてではありませんが）iOSやiPadOSで使うことができます。

たとえば「音声を読み上げるコマンド “say”**」を実行させるPythonコードが、次の例になります。

```
import os

os.popen('say "hello world”')
```

このPythonコードを実行すると、デフォルト設定である男性ボイスで、”ハロー・ワールド”と文字が読み上げられます。このように、macOSと同じようなコマンド機能も、iPhone/iPadで実は使えたりもします。

* https://www.matisse.net/OSX/darwin_commands.html

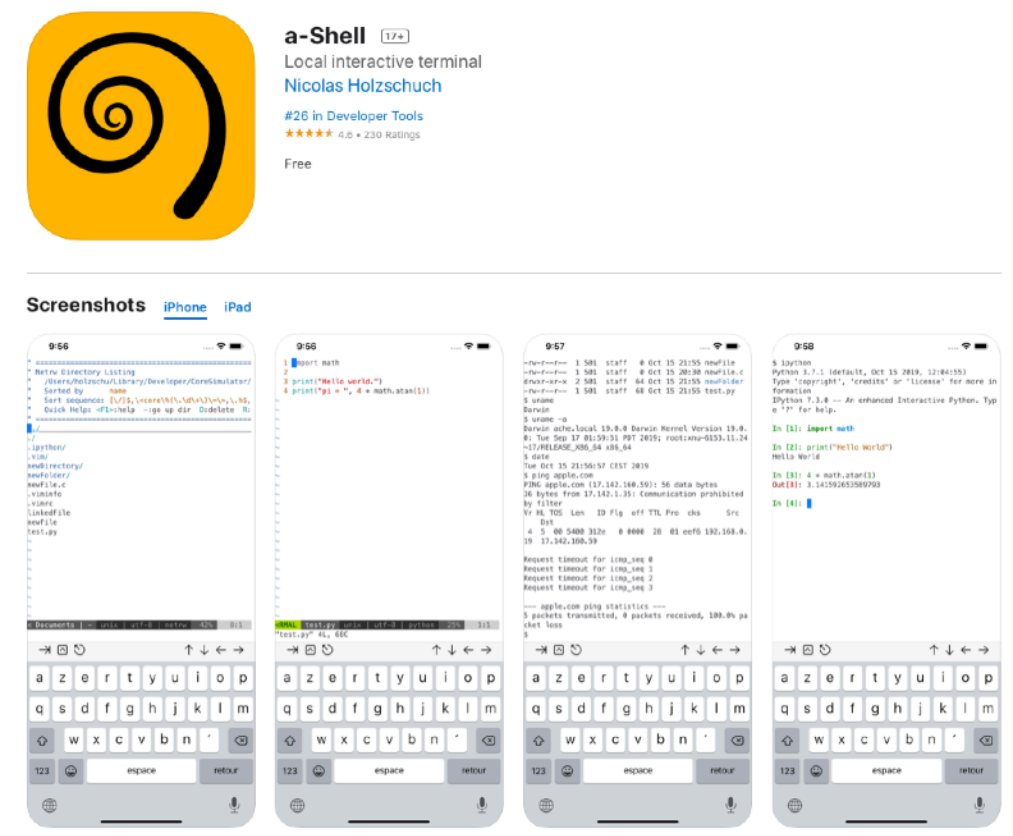
** https://1day1tip.yeno.net/2015/04/unix-command-say/

さらに詳しく使ってみたい人向け

ちなみに、Carnets for Jupyterの開発者が作り・App Storeで無料配布している a-Shellをインストールすると、iPhoneやiPadがターミナルベースのUNIXシステムに返信します。a-Shell上では、Lua, Python, JavaScript, C and C++といったプログラミング言語も使えますし、TeXやawkあるいはvimも…つまりはさまざまな環境がデフォルトでついてきます。

*https://holzschu.github.io/a-Shell_iOS/

https://apps.apple.com/jp/app/a-shell/id1473805438



1.3 各種フレームワークとObjective-Cランタイム

iPhoneやiPadを動かす基礎土台ともいうべきものがDarwinカーネルなら、その上で動く「本書ではとても重要な存在」が

・Objective-Cランタイム
・さまざまな機能を実現するフレームワーク（フレームワーク提供のAPI）

です。そのふたつについて、次は眺めていくことにしましょう。

1.3.1 Objective-Cランタイム

Objective-Cは、C言語の上に「Smalltalk言語に似たブジェクト指向記述」を持ち込んだ言語です。「クラスにはインスタンスがあって、クラスやインスタンスのメソッドがあって…」と書いていくと、C++などとほとんど同じように思われます。しかし、Objective-Cの場合、メソッド呼び出しについて「どの処理が呼び出されるか」は「実行時に動的に決定される」という点が、C++などとは大きく異なります。

そんな「実行時にどのメソッドを呼び出すかなど、実行時の各種処理をつかさどる」のが、Objective-Cランタイムです。

Objective-Cランタイムについて、Apple社の説明文言をそのまま使うと、「Objective-CランタイムはObjective-C言語の動的特性をサポートするためのランタイムライブラリ」というものになります。Objective-Cランタイムは、メッセージパッシングを介して（インタプリタのような）動的処理を実現しています。

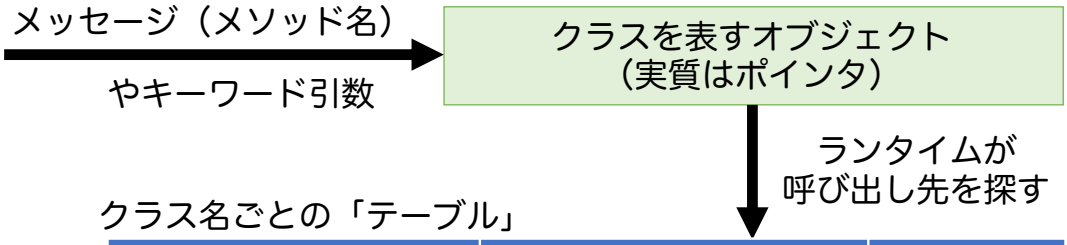
Objective-C
CをベースにSmalltalk型のオブジェクト指向機能を持たせた上位互換言語である。オブジェクトシステムはSmalltalkの概念をほぼそのまま借用したもので、動的型のクラス型オブジェクト指向ランタイムを持ち、メッセージパッシングにより動作する。実行時のメッセージパッシングの際、渡されるメッセージ値をセレクタという。
（https://ja.wikipedia.org/wiki/Objective-C）

たとえば、Objective-Cのクラスにメソッド実行をさせる時、そのクラス（やインスタンス）を表すオブジェクト（実質はポインタ）に対して、メソッド名やキーワード引数などに応じた「メッセージ」を送ります*。すると、Objective-Cランタイムは「（メソッド名やキーワード引数などに応じたメソッドを表す内部表現である）セレクタ」を見つけ出し、さらにセレクタをキーにして「呼び出しアドレス」をテーブルから決定し・その処理を実行させる……といった処理を、Objective-Cランタイムがつかさどります。

* 本書中でも、PythonからiOS/iPadOS機能を利用する際に、Objective-Cランタイムとのメッセージパッシングを Pythonから使います。

Objective-Cランタイムが担う「Objective-Cオブジェクト操作」は、次のようにして行われます。

- （プログラムコードから）オブジェクトに対して、メソッド名やキーワード引数を並べた「メッセージ」を送る
- Objective-Cランタイムが、クラス名ごとのテーブルから、呼び出すセレクタを決定する
- セレクタから「対応する関数の（呼び出しアドレス）」を決定する
- 関数呼び出しが実行される



クラス名ごとの「テーブル」

メソッド名 （C言語文字列）	セレクタ （メソッドを表す内部表現）	呼び出し アドレス
lockForConfiguration:	～	～
unlockForConfiguration	～	～
setTorchMode:	～	～

クラスのメソッドを呼び出したとき、「クラス名ごとのテーブルから、セレクタを介して動的に「呼び出す関数」が選ばれる」ということが、先に書いた「実行時に、どの処理が呼び出されていくかといったことが解決される」という内容です。

結局のところ「関数が呼び出しアドレスから実行」されるという「普通のC言語」的な処理ですが、その呼び出しアドレスが動的に決まるということが「C++などとは違う、というわけです。それが「Objective-Cは、C言語の上にSmalltalk言語に似たブジェクト指向記述を持ち込んだ言語です」ということになります。

つまるところ、Objective-Cのメッセージパッシングは「C言語形式の関数の呼び出し先が、条件に応じて適切に決まる」というものです。

1.3.2 さまざまな機能を実現するフレームワーク

「フレームワーク」は
・動的に読み込むことができる共有ライブラリ
・関連ファイル
をまとめたものです。Apple社が提供するフレームワークの数はとても多く、さらに、Apple社以外が提供するフレームワークも数多くあります。
iPhone/iPad上ではUNIX系のコマンド処理も可能ですが*、それらの機能だけで「やりたいこと」を実現するのはとても大変です。したがって、本書中で登場するiPhone/iPad機能をPythonから使うコード例では、その多くがフレームワーク機能を使った処理を行っています。

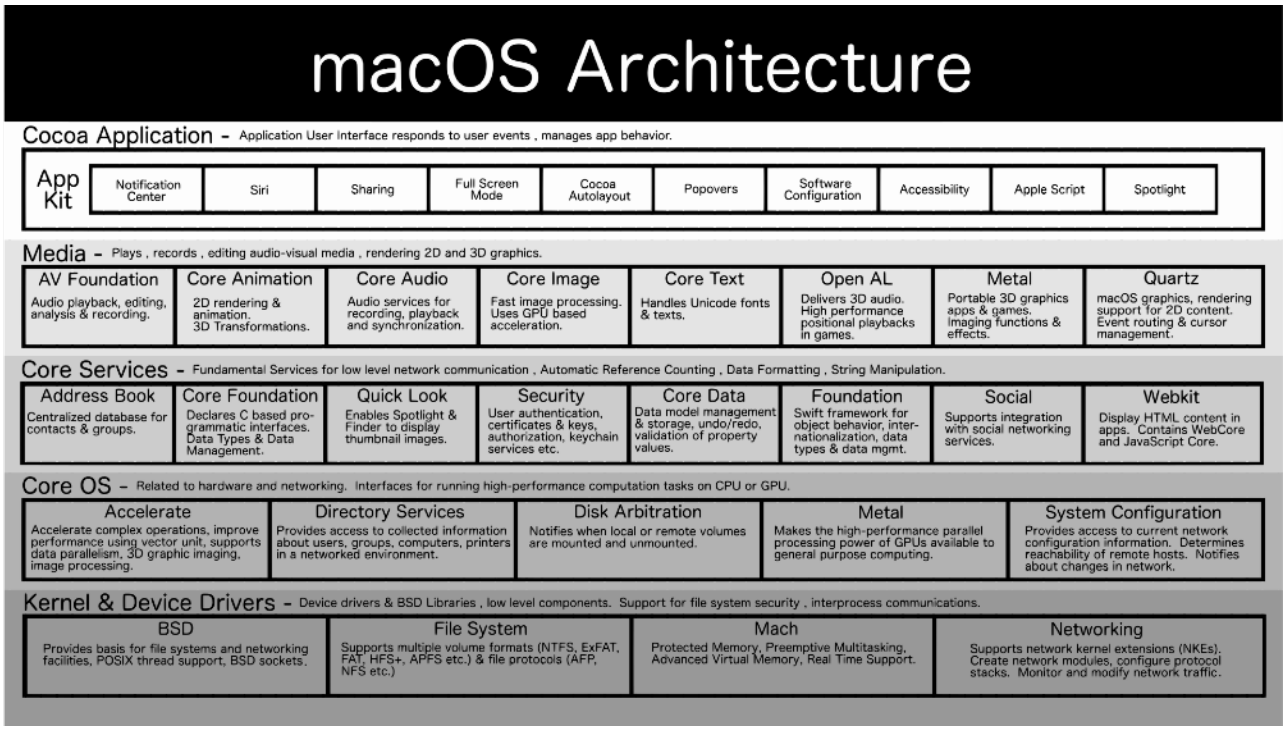
本書に登場する主要なフレームワークだけでも、画面表示（出力）や入力といったユーザインターフェイス関連の機能をまとめたUIKitフレームワーク、映像処理関連の機能を数多く備えるAVFoundationフレームワーク、デバイスが備えるセンサを使ってデバイスやユーザーの動きをこと細かく捉えることができるCore MotionフレームワークやCore Locationフレームワーク、機械学習に関する機能を備えたCoreMLフレームワークや画像処理関連機能をサポートするVisionフレームワーク、さらには拡張現実（AR)処理を簡単に行うことができるARkit ・・・と限りがありません。

次セクションでは、Apple社が提供しているフレームワーク機能を、Pythonから使ってみることにします。

2. iOSのフレームワーク関数をPythonから使う

2. iOSのフレームワーク関数をPythonから使う

iOSデバイス上でさまざまな処理を実現するために、iOS 上で使うことができる共有ライブラリは、フレームワーク(Framework)という形で提供されています。たとえば、下記は（iOSとかなりの部分で共通する）macOSで提供されているフレームワークなどを示した図です。



(https://commons.wikimedia.org/wiki/File:MacOS_Architecture.svg, Robryecran77 CC BY-SA 4.0)

iOS/iPadOSのフレームワークは、

・Publicフレームワーク

・Privateフレームワーク

と呼ばれる２種類のフレームワークに分かれます。Publicフレームワークは、公式に使うことが認められているもので、ドキュメントも整備されています。一方、PrivateフレームワークはApple社自身が使うためのフレームワークとして位置づけられているものです。

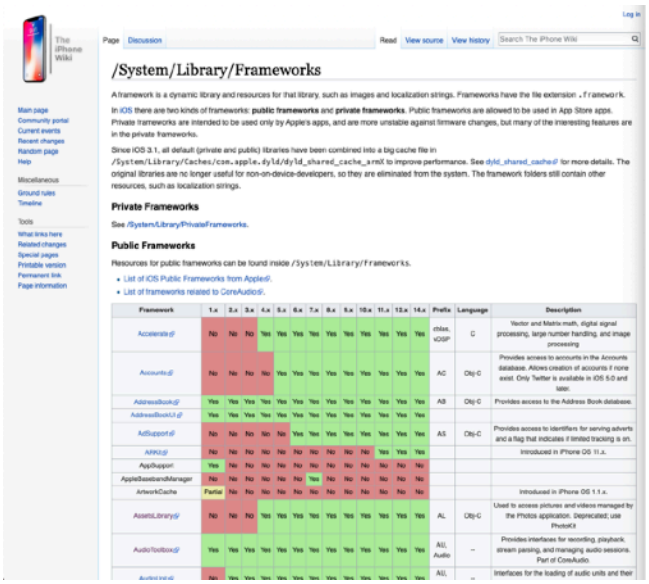
Privateフレームワークは、Apple社以外が使用することを想定していないため、Privateフレームワークを使ったアプリケーションは、App Store での配布が認められなかったりします。しかし、iOS上のPython環境で、Pythonコードから動的

に読み込み・使うのであれば、そうしたApp Storeでの配布制約を気にする必要はありません。Private フレームワークに含まれる機能でも、問題なく使うことができます。

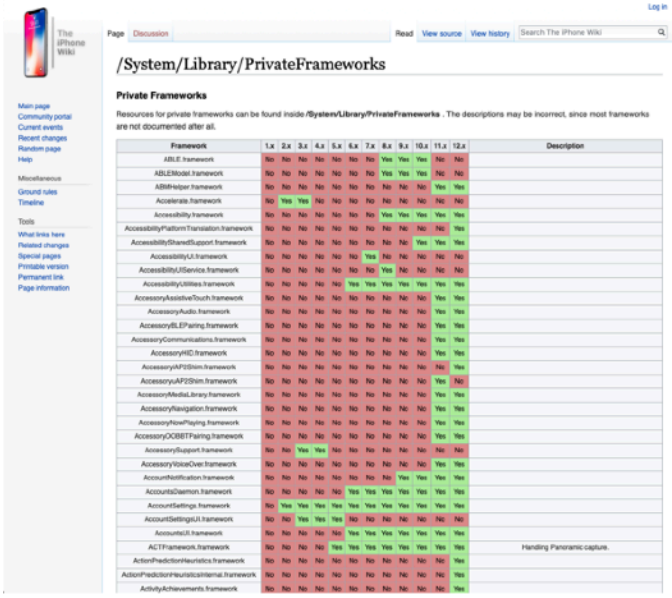
なお、フレームワークの一覧を確認したい場合には、iphonedev.wiki の紹介ページが役立ちます（Public: https://iphonedev.wiki/index.php/Frameworks, Private: https://iphonedev.wiki/index.php/PrivateFrameworks）。また、それぞれのフレームワークを使うことができるiOSバージョンについては、www.theiphonewiki.com にわかりやすくまとまっています。

Public：https://www.theiphonewiki.com/wiki//System/Library/Frameworks、
Private：https://www.theiphonewiki.com/wiki//System/Library/PrivateFrameworks

Publicフレームワークの一覧例



Privateフレームワークの一覧例



2.1 iOS のフレームワーク(共有ライブラリ)をctypesで使う

Apple社が提供するiOSの「フレームワーク」は動的に読み込むことができる共有ライブラリなどをまとめたものです。iOSを記述しているObjective-Cは基本的にはC言語ベースですから、PythonからC言語のライブラリを動的に読み込むためのcdll.LoadLibrary を使えば、iOSのフレームワークを読み込むことができます。そこで、まずは、Python に標準で付いてくるctypesライブラリを使って、iOSフレームワーク機能にアクセスしてみましょう。

ちなみに、cdll.LoadLibraryからiOSフレームワークを読み込む場合、iOS のセキュリティ制約から、cdll.LoadLibrary に引数として与える「フレームワークの場所」は「絶対パス」を与える必要があります。具体的には、次のような絶対パス、

- Public フレームワーク: /System/Library/Frameworks/〜.framework/〜
- Private フレームワーク: /System/Library/PrivateFrameworks/〜.framework/〜

を与えることになります。

2.2 Public フレームワーク関数を使う（サウンドを鳴らす）

手始めに、PublicフレームワークであるAudioToolBoxフレームワークを読み込んで、システムサウンドを再生してみます。

システムサウンド再生の手順は次の通りです。まず、ctypesパッケージを読み込んで、cdll.LoadLibraryを使ってAudioToolBoxを（フレームワークの場所を絶対パスで指定して）動的に読み込みます。次に、「システムサウンドを再生する」AudioServicesPlaySystemSound関数に対して「再生したいシステムサウンド種を引数として与えて呼ぶ」ことにより、iOSに備えられたシステムサウンドを鳴らすことができます。トータルでもわずか数行のPythonコードです。

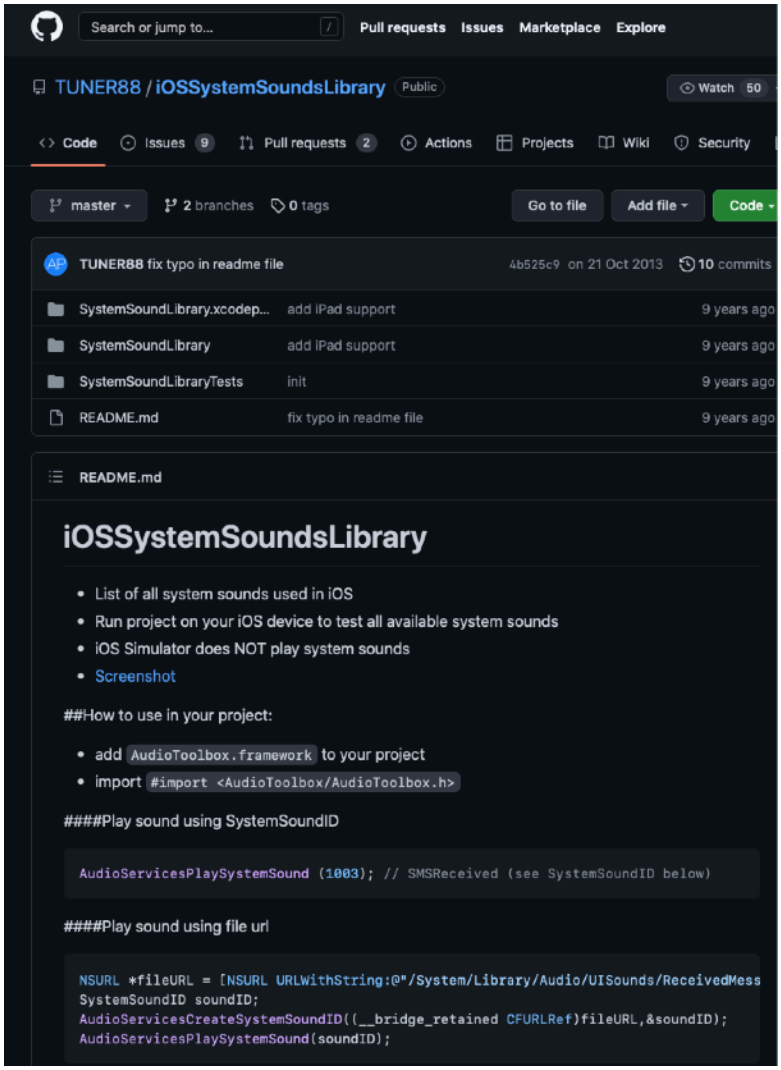
```
from ctypes import *

SIMToolkitGeneralBeep = 1052
```

```
    AudioToolbox = cdll.LoadLibrary(
      "/System/Library/Frameworks/AudioToolbox.framework/AudioToolbox"
    )
    AudioToolbox.AudioServicesPlaySystemSound( SIMToolkitGeneralBeep )
```

ちなみに、iOSで使うことができるシステムサウンドのリストは、iOSSystemSoundsLibrary（https://github.com/TUNER88/iOSSystemSoundsLibrary）にまとめられています。

iOSSystemSoundsLibrary
（https://github.com/TUNER88/iOSSystemSoundsLibrary）



List of SystemSoundIDs

SystemSoundID	File name	Category
1000	new-mail.caf	MailReceived
1001	mail-sent.caf	MailSent
1002	Voicemail.caf	VoicemailReceived
1003	ReceivedMessage.caf	SMSReceived
1004	SentMessage.caf	SMSSent
1005	alarm.caf	CalendarAlert
1006	low_power.caf	LowPower
1007	sms-received1.caf	SMSReceived_Alert
1008	sms-received2.caf	SMSReceived_Alert
1009	sms-received3.caf	SMSReceived_Alert
1010	sms-received4.caf	SMSReceived_Alert
1011	-	SMSReceived_Vibrate
1012	sms-received1.caf	SMSReceived_Alert
1013	sms-received5.caf	SMSReceived_Alert
1014	sms-received6.caf	SMSReceived_Alert
1015	Voicemail.caf	-
1016	tweet_sent.caf	SMSSent
1020	Anticipate.caf	SMSReceived_Alert
1021	Bloom.caf	SMSReceived_Alert
1022	Calypso.caf	SMSReceived_Alert
1023	Choo_Choo.caf	SMSReceived_Alert
1024	Descent.caf	SMSReceived_Alert
1025	Fanfare.caf	SMSReceived_Alert
1026	Ladder.caf	SMSReceived_Alert
1027	Minuet.caf	SMSReceived_Alert
1028	News_Flash.caf	SMSReceived_Alert
1029	Noir.caf	SMSReceived_Alert
1030	Sherwood_Forest.caf	SMSReceived_Alert
1031	Spell.caf	SMSReceived_Alert
1032	Suspense.caf	SMSReceived_Alert
1033	Telegraph.caf	SMSReceived_Alert
1034	Tiptoes.caf	SMSReceived_Alert
1035	Typewriters.caf	SMSReceived_Alert
1036	Update.caf	SMSReceived_Alert
1050	ussd.caf	USSDAlert
1051	SIMToolkitCallDropped.caf	SIMToolkitTone
1052	SIMToolkitGeneralBeep.caf	SIMToolkitTone

2.3 Private フレームワーク関数を使う（数式処理を使う）

次は、Privateフレームワークの関数を、Pythonから使ってみます。Apple 社のみが使うことを想定しているのが、Private フレームワークです。したがって、フレームワークに関する公式解説ドキュメントは提供されていません。そのため、Privateフレームワークを使う場合、フレームワークのバイナリファイルからヘッダーファイルを生成・公開しているサイト情報*などを頼りに、フレームワーク機能を使うことになります。

*ヘッダーファイル群を分析して公開しているサイトの例:

- https://developer.limneos.net/index.php
- https://github.com/mmmulani/class-dump-o-tron
- https://gist.github.com/githubutilities/dfa762a11e1064bfce5f

詳しくは https://code.google.com/archive/p/undocumented-goodness/source/default/source

ここでは、iOS の Private フレームワークのひとつ、Calculateフレームワークを Pythonから試しに使ってみます。Calculateフレームワークは、その名の通り「計算を行うためのフレームワーク」です。フレームワークに含まれる関数はただひとつ、

・文字列を引数として渡されると、その文字列をもとに数式計算する

という CalculatePerformExpression 関数です。

CalculatePerformExpression 関数の引数や返り値は、フレームワークを分析してヘッダーファイルを公開しているサイト情報を信頼すると、

CalculatePerformExpressionの int CalculatePerformExpression(char *expr, int significantDigits, int flags, char *answer);

という定義です。そこで、

・文字列として与える数式：expression
・計算結果を格納するための文字列：result

を用意した上で*、

・有効桁数：significantDigits
・計算条件フラグ**: flag
を適当に与えてやると、数式文字列に対する計算結果を見事に得ることができます（計算できた場合には、success に1が返されます）。

*CalculatePerformExpression との「char *　としての文字列」受け渡しについては、UTF-8 符号化による文字列化(char * 化)が必要なので、encode(“utf-8”) と decode() でPython 3 Unicode文字列への変換をしています。

**計算条件フラグは、列挙型 CalculateFlags として下記の定義のようです。

・ CalculateUnknown1 = 1 << 0 = 1,

・CalculateTreatInputAsIntegers = 1 << 1 = 2,

・CalculateMoreAccurate = 1 << 2 = 4

```
from ctypes import *

Calculate = cdll.LoadLibrary(
  "/System/Library/PrivateFrameworks/Calculate.framework/
Calculate"
)
expression = “pi * sin(2 / pi)”
result = create_string_buffer( 1024 )
success = Calculate.CalculatePerformExpression(
    expression.encode("utf-8"), 16, 1, result)
print( "Result is : %d (success=1, fail=0), %s"
      %(success, result.value.decode()) )
```

このように、iOSのフレームワーク、つまり動的な共有ライブラリをcdll.LoadLibraryを使って動的に読み込むことで、iOSのフレームワークに含まれる関数を使うことができます。

Publicフレームワークを使うことができるのは当然ですが、Privateフレームワークであっても、Pythonコードから動的に使うことになります。

内容については正式版を参照ください。

https://techbookfest.org/organization/u6z4wFyQP5Y8HYyv9CKayn

2.4 Objective-Cクラスをctypesモジュールで使う

前セクションでは、iOSのフレームワーク(Public/Private)を動的に読み込み、フレームワークに含まれる関数を使ってみました。このように、フレームワーク内の関数を使うことができるのは便利ですが、フレームワークに含まれるのは関数だけではありません。それ以外にも、さまざまな機能を実現するために作られたクラスも含まれています。

そこで次は、ctypes パッケージを使って、Python からiOSフレームワークが提供するクラス機能にアクセスしてみることにします。

・AVFoundation: フラッシュライトを制御する

それでは、cdllライブラリを介してObjective-Cクラス機能を使ってみます。扱う題材は「AVFoundationが提供するフラッシュライトの制御機構」です。ctypesライブラリの機能を使ってObjective-Cクラスとメソッドを呼び出して、フラッシュライト制御をしてみます*。

> ＊後述するRubicon-objパッケージや（過去のRubicon-objをもとにした）Objc_Utilモジュールも、内部で行っているのはctypesモジュールを使った処理です。したがって、本例を眺めておけば、Rubicon-objやObjc_utilが行っている内容を想像することができることでしょう。そしてまた、Rubicon-objやObjc_utilを読んだ時に「なるほど、そうやってるのね」といった理解もしやすくなるはずです。

サンプルコード：_carnets_torch.ipynb
ライブラリコード：avfoundation/torch.py

まずは、ctypes.cdll.LoadLibraryで「引数をNoneとする」ことで、アプリ実行時に読み込まれているライブラリ群を読み込みます。そして、Objective-Cランタイムの機能である

・objcクラス名からクラスオブジェクトへのポインタを得る関数
”objc_getClass” を使う

ため、”objc_getClass” 関数の引数や返り値の型を（.argtypesや.restypeを使って）設定しています。これは、Ctypes.cdll.LoadLibraryでライブラリを読み込む

だけでは、関数アドレスに対するポインタは得られても、引数や返り値の型に関する情報が得られないからです。そのため、このように型情報を設定してやる必要があります。

ちなみに、cls()はobjc.objc_getClass()を短くするため作成した、ヘルパー関数です。

```
from ctypes import c_void_p, c_char_p, c_int, c_bool, cdll

# ランタイムを読み込む
objc = cdll.LoadLibrary(None)
# objcクラス名からクラスオブジェクトへのポインタを得る"objc_getClass"の型設定
objc.objc_getClass.argtypes = [c_char_p]
objc.objc_getClass.restype = c_void_p

# objcクラスへのポインタを得るヘルパー関数
def cls(cls_name):
    return objc.objc_getClass(cls_name)
```

次に、メソッド名やラベル（引数）情報からセレクタ（へのポインタ）を得る"sel_registerName"関数に対して、引数や返り値の型設定をします。

```
# メソッド名などからセレクタを得る"sel_registerName"の型を設定
objc.sel_registerName.restype = c_void_p
objc.sel_registerName.argtypes = [c_char_p]
```

ここで「セレクタを得る」必要があるのは次の通りです。短く書くと、「Objective-Cランタイムに"このオブジェクトにこのメソッドを実行してよ"というメッセージを投げるときに、メソッド種をセレクタで指定する」からです。

そして、Pythonからオブジェクトにメソッド（セレクタ）を投げるヘルパー関数も作っておきます。

```
# メッセージを投げて結果を得るためのヘルパー関数
def msg(obj, restype, sel, argtypes=None, *args):
    if argtypes is None:
```

```
            argtypes = []
        # objc_msgSendの引数・返値型を与える（引数の先頭２つは、obj, メソッド名）
        objc.objc_msgSend.argtypes =  [c_void_p, c_void_p] + argtypes
        objc.objc_msgSend.restype = restype
        # メッセージを投げる
        res = objc.objc_msgSend(obj,
                                objc.sel_registerName(sel),
                                *args)
        return res
```

そして、準備としては最後の仕上げとして、Objective-Cの文字列とPython 3文字列間の違いを吸収するために、UTF8文字をNSStringに変換する関数も作っておきます。この関数実装では、基本的なクラスなどを提供するFoundationフレームワークのNSStringクラスが持つメソッド ”stringWithUTF8String:” を使って、「NSStringクラスに、クラスメソッドであるstringWithUTF8String:を実行して！とメッセージを投げることで、UTF-8文字（へのポインタ）をバイト列sに変換する」という処理を実現しています。

ここまでで、準備は完了です。

```
    # UTF-8のバイト列文字をNSStringにして返す関数
    def nsstr(s):
        return msg( cls(b'NSString'),
                    c_void_p,                   # 返り値型
                    b'stringWithUTF8String:',
                    [c_char_p],                 # 引数型
                    s)                          # バイト列
```

さて、ここからが本番です。Objective-Cランタイムにメッセージを投げて「AVCaptureDeviceクラスからビデオ撮影用のカメラデバイスを取得」して、さらにメッセージを投げて「ライト機能が備えられているかを確認」して、ライト機能を使うことができるようであれば、やはりメッセージを投げて、

- On (=1)
- Off (=0)

として「点灯状態（setTorchMode）を設定」する……そんな処理をヘルパー関数にしてみた例が、次のコードです。

```
    # ライトの点灯・非点灯状態を指定する
    def setTorchMode(state=0): # 1:turn on, 0 turn off
        # AVCaptureDeviceクラスへのポインタを得る
        AVCaptureDevice = cls(b'AVCaptureDevice')
        # AVCaptureDeviceに引数'vide'で
        # メッセージdefaultDeviceWithMediaType:を投げて、デバイスを取得
        device = msg( AVCaptureDevice, c_void_p,
                      b'defaultDeviceWithMediaType:',
                      [c_void_p], nsstr(b'vide'))
        # 得たデバイスにライトがあれば(hasTorch=True)
        has_torch = msg(device, c_bool, b'hasTorch')
        # ライトがなければランタイムエラーを出す
        if not has_torch:
            raise RuntimeError(b'Device has no flashlight')
        # Configurationをロックして、点灯状態を設定する
        msg(device, None, b'lockForConfiguration:', [c_void_p], None)
        msg(device, None, b'setTorchMode:', [c_int], state)
        msg(device, None, b'unlockForConfiguration')
```

なお、AVCaptureDeviceの設定を変える場合、排他的な処理をするために、

・lockForConfiguration:
・unlockForConfiguration

というふたつの関数を「AVCaptureDeviceの設定を変える」前後で呼び、デバイス設定を「ロック」したり「ロック解除」したりする必要があります。

ここまでに書いたヘルパー関数を使って、「ライトを付けて、1秒後にライトを消す」処理を行うPythonコード例が下記になります。

```
    import time
```

```
    setTorchMode(1) # On
    time.sleep(1)
    setTorchMode(0) # Off
```

このように、C言語で書かれた関数を使うことができるctypesパッケージを読み込み、「Objective-Cランタイムに対してメッセージを投げる」ことにより、C言語形式の関数呼び出しを介して、PythonからObjective-Cのクラスを使うことができました。

しかし、その作業を実現するために「たくさんのヘルパー関数を書く」必要がありました。これでは少し面倒です。もっと手軽にObjective-CクラスをPythonから操りたいものです。

そこで、次は、Rubicon-ObjCや（過去バージョンのRubicon-ObjCをもとにした）objc_utilを使って、Objective-Cクラスやランタイムを簡単に使うためのプログラミング手順を眺めていくことにします。言い換えれば、「ctypesなどを使って書かれた、PythonからObjective-Cクラスを使う/ランタイムにアクセスするための便利な関数群」を使って、iOSデバイス・プログラミングを実現するPython コードを書いてみることにします。

ポイント

- Objective-CのベースはC言語、C言語形式の関数呼び出しを行なって、Objective-Cランタイムとやりとりすれば、Objective-Cクラスを使いこなすことはできる
- とはいえ、自前で全部書いていくのは「かなり面倒」
- そんな時はRubicon-ObjCやObjc_utilを使おう

内容については正式版を参照ください。

https://techbookfest.org/organization/u6z4wFyQP5Y8HYyv9CKayn

3. PythonとObjective-Cを橋渡しするパッケージ

3. PythonとObjective-Cを橋渡しするパッケージ

Objective-Cランタイム機能をPythonから簡単に使う方法としては、

・Rubicon-ObjCパッケージを使う

・Objc_utilモジュールを使う

という、ふたつの方法があります*。

Rubicon-ObjCパッケージは、「ctypesライブラリなどを使い、Objective-Cランタイムとのやりとりを隠蔽する関数群を使うことで、Objective-CクラスをPythonのクラスとして簡単に扱うことができるようにしたパッケージ」です。PythonistaやPytoでは、アプリをインストール時点ですでにRubicon-ObjCがインストールされています。また、CarnetsやJunoでは、Python Package Index (PyPI)を使ってインストールすることができます。

そして、Objc_utilモジュールは、過去バージョンのRubicon-ObjCをもとに作られたモジュールで、Pythonista向けに作られ・Pythonistaに添付されている（別途配布もされている）ものです。Pythonista特有のUI機能など、一部機能が使えないことをのぞけば、他のPython環境アプリでも多くの機能を使うことができます。

なお、本書（前編・後編）で使う・作るパッケージでは、Rubicon-ObjCパッケージや（Pythonista環境に依存しないようにした）objc_utilモジュールを使っています。手順にしたがって本書のサンプルファイルをダウンロードして使う場合には、Rubicon-ObjCパッケージやPythonista非依存のobjc_utilモジュールを添付しているため、追加でパッケージやモジュールをインストールする必要はありません。

* PythonとObjective-Cの間を「橋渡し」するライブラリには、他にPyObjC (https://pyobjc.readthedocs.io/en/latest/)というものもあります。しかし、PyObjCが現状でサポートするのは「macOSのみ」なので、本書中では取り扱いません。

3.1 PythonとObjective-Cを橋渡しするRubicon-objc

「ctypesなどを使って書かれた、PythonからObjective-Cクラスを使う/ランタイムにアクセスするための便利な関数群」として、Rubicon-objcと（過去バージョンのRubicon-objcをもとにした）Objc_utilがあります。

まず最初に、Rubicon-objc (https://pypi.org/project/rubicon-objc/) の使い方を知っておきましょう。

Rubicon-objC は、BeeWare (https://docs.beeware.org/en/latest/index.html) というライブラリ・ツール集に含まれている、PythonからiOSやmacOSのObjective Cライブラリを使うことができるライブラリです*。BeeWareに含まれる他ライブラリとしては、iOSを含むクロスプラットフォームでGUIアプリケーションを作ることができるToga (https://beeware.org/project/projects/libraries/toga/)などがあります。

Rubicon マニュアル:

https://rubicon-objc.readthedocs.io/_/downloads/en/latest/pdf/

How to guide:

https://rubicon-objc.readthedocs.io/en/latest/how-to/index.html

3.1.1 Rubicon-ObjCのインストール

本書の前編（活用編）で使った「お手軽3分クッキング」的なパッケージを導入している方は、すでに、作業ディレクトリにRubicon-ObjCがインストールされています。したがって、下記作業は不要です。その上で、「お手軽3分クッキング」的なパッケージをインストールしていない方のため、あるいは今後のために、Rubicon-ObjCのインストール方法を書いておきます。

まず、PythonistaやPytoを使っている場合には、標準でRubicon-ObjCがインストールされています。そのため、新たにRubicon-ObjCをインストールする必要はありません。CarnetsやJunoを使っている場合であれば、初期状態ではインストールされていないため、次の手順でRubicon-ObjCをインストールします*。

内容については正式版を参照ください。

https://techbookfest.org/organization/u6z4wFyQP5Y8HYyv9CKayn

4. iOS/iPad機能を
ネイティブに使う「レシピ集」

4. Rubicon-ObjCやObjc_utilを使うレシピ大全集

それでは、PythonからObjective-Cクラスを簡単に扱うことができるRubicon-ObjCやObjc_utilを使って、場合によってはctypesだけを使って、さまざまなフレームワーク機能をPythonから操ってみることにしましょう。

基本的にはフレームワークごとに、機能実装のレシピをさらいながら、ついでに

- Objective-C APIを使うコード書きのポイント
- Objective-C APIコードの書きにおけるデザインパターン

をさらっていきます。

それでは、ページをめくり、Pythonで各種フレームワークを操るための「レシピ集」「コードテクニック大全集」を眺めていきましょう。

注意：本書冒頭で使った3分クッキング的パッケージでは、（名前空間を別にするために）rubicon-objcをrubicon_objc というフォルダに入れています。そのため、 読み込むためには、import rubicon_objc とする必要があります。rubicon-objcを通常の手順でインストールでした場合には、import rubicon.objc といったように（コード例を）読み替えて下さい。

●Assets Library

まずはじめは、画像や動画を扱うためのAssetsライブラリーです。Assetsライブラリーは、すでにiOS9以降では非推奨となっていて、iOS8から使われているPhotoKit フレームワークを使うことが求められています。そのため、最近の機種では「そんな機能無いよ」と言われたりもしますが、とても面白い機能を備えている（備えていた）ので、まずはここから始めてみます。

・画像からアスキーアートを生成する

次のコードは、Rubicon-ObjCを使い、iOS 12.1 までは実装されてた「画像からアスキーアート」を生成する例です。処理内容は、UIImageクラスの「ファイルからUIImageを生成・初期化」するinitWithContentsOfFile_で、UIImageに画像を読み込みます。そして、UIImageの（iOS 12.1までは実装されていた）

・ASCIIDescriptionWithWidth_height_(横ピクセル数, 縦ピクセル数)

を呼ぶと、（ASCII文字で画像を表現する）アスキーアートを画像から生成して、その文字列が返されます。その文字列をプリントすれば、昔懐かしいアスキーアートができあがる、というわけです。

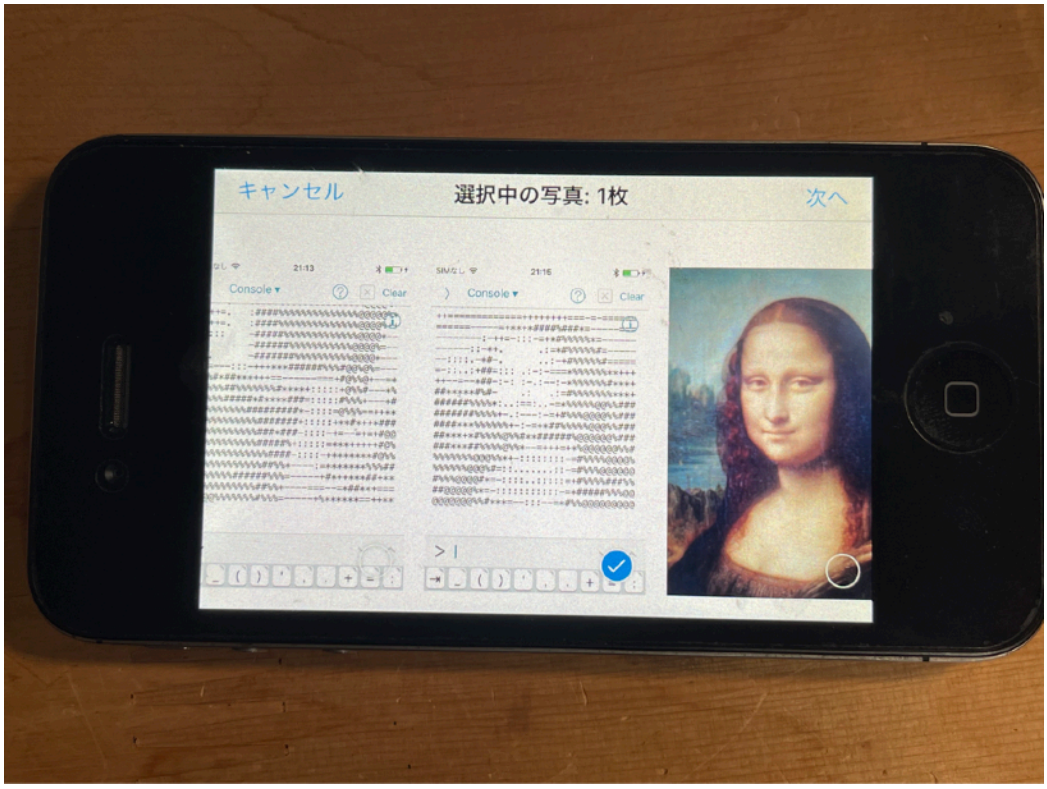
```
from os.path import abspath
from ctypes import *
from rubicon.objc import ObjCClass

AssetsLibrary = cdll.LoadLibrary(
"/System/Library/Frameworks/AssetsLibrary.framework/AssetsLibrary"
)
#  UIImageクラスのインスタンスを生成
UIImage = ObjCClass("UIImage")
image = UIImage.alloc().initWithContentsOfFile_("Mona_Lisa.jpg")

# iOS 12.1 までならこの関数を使うことができる
aa = image.ASCIIDescriptionWithWidth_height_(
    int(image.size.width/10), int(image.size.height/10) )
```

```
    print( aa )
```

　このコード（あるいはセル）を実行すると、次のような文字で画像を描いたアスキーアートが出力されます。



　ちなみに、バーストモードで撮影されたファイルを読み込み処理させると、アニメーションGIFが生成されます。その場合のコードは、以下のようになります。

c.f. http://hitoriblog.com/?p=41049

ポイント
- ObjCClass(“クラス名")でクラスを生成
- クラス.alloc().init〜　でクラスインスタンスを生成
- クラス.〜　や クラスインスタンス.〜で、メソッドにアクセスする

●UIKit フレームワーク

UIKitフレームワークは、ユーザーインターフェース（UI）に関係する幅広い領域の機能を提供しています。たとえば、画面に表示されるウィンドウやさまざまな部品、あるいは画面入力やアニメーション描画や画像機能。UI関連の膨大な機能を提供しているのが、iOS2.0から使うことができるUIKitフレームワークです。

• バッテリー残量など、各種デバイス情報を得る

UIKit フレームワークに含まれるUIDeviceクラスを使うと、バッテリー残量やOSバージョン、あるいは、デバイスの向きといった、iOS/iPad OSデバイスに関するさまざまな情報を得ることができます。 UIDeviceから得ることができる情報としては、たとえば、

- name： デバイスの名前
- systemName：OSの名前
- model：デバイスのモデル
- localizedModel：ローカライズされたデバイスのモデル
- orientation：デバイスの向き

などがあります*。

* https://developer.apple.com/documentation/uikit/uidevice?language=objc

次のコード例は、UIKitフレームワークを読み込み、Rubicon-ObjCを使ってUIDevice Objective-Cクラスのプロパティにアクセスすることで、各種情報の表示を行う例です。

c.f. : / uikit / ui_device.py (Notebookは_carnets_ui_device_battery.ipynb)

```
import ctypes
import rubicon_objc

UIKit = ctypes.cdll.LoadLibrary(
  "/System/Library/Frameworks/UIKit.framework/UIKit"
)
```

```
    UIDevice = rubicon_objc.api.ObjCClass( 'UIDevice' )
    UIDevice_currentDevice_ = UIDevice.currentDevice

    print( UIDevice_currentDevice_.batteryLevel )
    print( UIDevice_currentDevice_.orientation )
    print( UIDevice_currentDevice_.systemVersion )
    print( UIDevice_currentDevice_.name )
    print( UIDevice_currentDevice_.systemName )
    print( UIDevice_currentDevice_.model )
    print( UIDevice_currentDevice_.localizedModel )
    print( UIDevice_currentDevice_.orientation )
```

この例ではRubicon-ObjCを使いましたが、Objc_utilを使って書く場合には、プロパティ部分について、たとえばbatteryLevelの部分はbatteryLevel()といった書き方になります。

ポイント

- Rubicon-ObjCではクラスのプロパティに()はつけない
- Objc-utilではクラスのプロパティに()をつける

・画面の明るさを設定したり・明るさを調べる

次は、Objc-utilを使い、画面（ディスプレイ）の明るさを得たり・画面（ディスプレイ）の明るさを設定してみます。Objc-utilを使っている理由は、「UIKitに関する情報更新はメインスレッドで行う」という決まりがあり、Objc-utilでは@on_main_threadというキーワードをつけることで、「メインスレッドでの処理にする」と指定できるからです。

サンプルファイル：carnets_uikit_devicescreen_brightness_get_and_set.ipynb
ライブラリーファイル：uikit/ui_screen_brightness.py

画面の明るさを得るために、画面に関する情報をつかさどるUIScreenクラスから、デバイスのメイン画面を表すクラスプロパティのmainScreen*にアクセスし、さらにmainScreenのbrightnessプロパティにアクセスすることで、「画面の明るさ」に関する情報を読み込んだり・書き込んだりします。

*https://developer.apple.com/documentation/uikit/uiscreen/1617815-mainscreen

iOS 16までしかサポートがされないようです。

```
from objc_util import *

@on_main_thread
def get_screen_brightness():
    UIScreen = ObjCClass('UIScreen')
    return float(UIScreen.mainScreen().brightness())

@on_main_thread
def set_screen_brightness(brightness):
    UIScreen = ObjCClass('UIScreen')
    UIScreen.mainScreen().brightness = brightness
```

このコードでは、mainScreenのbrightnessプロパティに対して、

- 読み出し時：UIScreen.mainScreen().brightness()

● 書き込み時：UIScreen.mainScreen().brightness

というようにアクセスをしています。プロパティから値を読みこむときはプロパティ名に()を付けますが、値を書き込む際にはプロパティ名に()を付けません。Objc-Utilがプロパティへのセッターを呼ぶことで、プロパティに対する値設定が行われます。

ポイント

● UIKitの情報更新はメインスレッドで行う（Objc-Utilを使うと便利）

● Rubicon-ObjCでの「プロパティへの値書き込み」時には()をつけない

• 触覚フィードバックを制御する

Rubicon-ObjCを使い、UIImpactFeedbackGeneratorクラスのインスタンスを生成して、インスタンスのメソッドを呼ぶことで、触覚（Haptics）フィードバックを行う例です。ここでは、ObjCClass('UIImpactFeedbackGenerator')で、クラスオブジェクト（のポインタ）を得た上で、.alloc().init()とすることで、Objective-Cクラスのインスタンスを生成して、.prepare()や.initWithStyle()、あるいは.impactOccurred()といったメソッドを呼んで、触覚フィードバックの設定と振動生成を行なっています。

サンプルファイル：ui_impact_feedback_generator.py
ノートブック：_carnets_uikit_impact_feedback_generator.ipynb

```
from rubicon.objc import ObjCClass

# クラスオブジェクトへのポインタを得る
UIImpactFeedbackGenerator = ObjCClass('UIImpactFeedbackGenerator')
# インスタンス作成
aUIImpactFeedbackGenerator =
UIImpactFeedbackGenerator.alloc().init()
aUIImpactFeedbackGenerator.prepare()

style = 0 # 0-4
aUIImpactFeedbackGenerator.initWithStyle_(style)
# 触覚フィードバック生成
aUIImpactFeedbackGenerator.impactOccurred()
```

このコード中の

- UIImpactFeedbackGenerator_.initWithStyle_(style)

は、

- UIImpactFeedbackGenerator_.initWithStyle(style)

と書くこともできます。前者は「メソッド名やラベル名のコロン(:)をアンダースコア(_)に書き換えて並べて書くFlat記法」と解釈され、後者は「ラベル名:値”をキーワード引数で”ラベル名=値”として書くInterleaved記法」と解釈されるからです。このメソッドは、Objective-Cで書くと*

● initWithStyle:(UIImpactFeedbackStyle)style;

というように、ラベル名が使われていないメソッドです。そのため、Flat記法とInterleaved記法で書いた差が、メソッド名の後にアンダーバー”_”が付くか・付かないかだけになるわけです。つまり、ラベル名がないメソッドはInterleaved記法で書くと、タイプ数を（一回だけですが）減らすことができます。

＊https://developer.apple.com/documentation/uikit/uiimpactfeedbackgenerator/2374286-initwithstyle

ポイント

● ラベル名がないメソッドはFlat記法・Interleaved記法ほとんど同じ

● ラベル名がないメソッドはInterleaved記法で書くと楽

内容については正式版を参照ください。

https://techbookfest.org/organization/u6z4wFyQP5Y8HYyv9CKayn

5. Xcodeでビルドして AppStoreで配布する

内容については正式版を参照ください。

https://techbookfest.org/organization/u6z4wFyQP5Y8HYyv9CKayn