Projekty sprzętowe – bo nie tylko webem człowiek żyje

Rozwój urządzeń mobilnych i postępująca cyfryzacja naszego życia, czy popularyzacja bezprzewodowego dostępu dostępu do Internetu i bezprzewodowej komunikacji otaczających nas urządzeń tworzą nowe rynki, na których twórcy i ich startupy odnoszą sukcesy oferując nowe urządzenia i usługi. Projekty sprzętowe otwierają przed nami nowy segment rynku, czy też stanowią przyjemną odskocznię od np. webowej rutyny.

Żeby skryptować elektronikę i tworzyć nowe urządzenia, czy sprzętowe usługi nie trzeba już znać się na programowaniu mikrokontrolerów. Nie trzeba wkuwać wielkich ksiąg poświęconych poszczególnym układom. Rozwój narzędzi deweloperskich pozwala nam już skryptować elektronikę z poziomu Pythona, czy nawet JavaScriptu. Czego więc potrzebujemy by zacząć przygodę ze sprzętowymi projektami?

Pierwsze kroki i prototypowanie

Osoby stawiające pierwsze kroki ze skryptowaniem elektroniki zapewne natrafią na Raspberry Pi, czy Arduino. Są to obecnie chyba najpopularniejsze marki w branży *sprzętowej*.

Raspberry Pi to mini komputer, na którym po uruchomieniu Linuksa możemy tworzyć np. aplikacje desktopowe zintegrowane z peryferiami USB, czy też tymi podłączonymi za pomocą pinów GPIO. Dostępne na rynku mini komputery (a jest ich wiele) pozwalają tworzyć nowe urządzenia i usługi integrujące biblioteki i możliwości komputera z elektroniką taką jak czujniki, wyświetlacze, czy różnego rodzaju silniki.

Arduino to przedstawiciel mikrokontrolerów. Płytka Arduino możemy przypominać Raspberry, ale nie dostajemy na niej procesora, tylko znacznie prostszy mikrokontroler. Za pomocą dostarczanego SDK i edytora jesteśmy w stanie zaprogramować mikrokontroler na płytce. Zazwyczaj oznacza to że np. odpada nam obsługa peryferiów USB, czy możliwość wykorzystania bibliotek znanych z Linuksa. Z drugiej strony mikrokontrolery są znacznie bardziej energooszczędne i tańsze (nawet od Raspberry Pi). Świetnie sprawdzają się w projektach, w których musimy zasilać układ z baterii przez długi okres czasu, czy gdzie ważną rolę odgrywają koszty (a zastosowanie komputera nie jest potrzebne).

Na rynku dostępnych jest wiele mikrokontrolerów i mini komputerów. Nie wszystkie oferują biblioteki i API dla Pythona. Spośród mini komputerów przyjazne dla Pythona jest np. Raspberry Pi, Beaglebone Black, czy PcDuino. W przypadku mikrokontrolerów nadal króluje C (Arduino), czy C++ (mbed), ale Python też zaczyna coraz częściej zaznaczać swoją obecność. MicroPython (pyboard), czy PyMCU to płytki skryptowane z poziomu Pythona. PyBoard posiada

własną (ograniczoną) implementację Pythona na mikrokontrolery, natomiast PyMCU jest skryptowane z poziomu Pythona na komputerze, do którego jest podłączona. Także Arduino w pewnym zakresie można skryptować Pythonem (za pomocą PyFirmata i nakładek). Dodatkowo wymienić można projekty takie jak TinkerForge, Phidgets, czy Synapse Snap.

Jeżeli **chcesz zacząć** przygodę z elektroniką to polecam wybranie jakiegoś zestawu startowego. Nauka to nic innego jak budowanie kolejnych prostych układów i skryptowanie ich. Czy to będzie miganie diodą LED, poruszanie serwomechanizmami, silnikami krokowymi, albo DC, czy też wykorzystanie wyświetlaczy LCD, komunikacja szeregowa z innymi złożonymi układami. Jak opanujesz API danej platformy będziesz w stanie łączyć działanie poszczególnych elementów ze sobą. Np. stworzyć samojezdnego robota, który będzie unikał przeszkód dzięki czujnikom zbliżeniowym.

Co do **startowej platformy** to możliwości jest co najmniej kilka. Dla PyMCU opublikowałem już szereg artykułów. PyBoard zapewne też niebawem stanie się popularny (bo m.in. może działać bez podłączenia do PC). Arduino z poziomu Pythona nie oferuje wszystkich możliwości, ale proste układy da się w nim obsłużyć (biblioteka BreakfastSerial). Pośród mini PC mamy Raspberry Pi, ale także Beaglebone Black, czy PcDuino - choć do tych dwóch ostatnich liczba tutoriali i przykładów będzie mniejsza niż w przypadku Raspberry Pi. Ich zalety to np. większa ilość pinów GPIO, czy silniejszy procesor.

Rozwiązania praktyczne

W pewnym momencie będziesz chciał przejść z poziomu prototypu na płytce stykowej do bardziej praktycznej wersji swojego układu. W zależności czy to ma być jeden układ, czy produkcja masowa dla szerszej grupy odbiorców czeka Cię szereg kroków jakie należy wykonać by osiągnąć wyznaczony cel. Nikt nie kupi Twojego urządzenia jeżeli będzie drogie, czy nieporęczne (bo np. siedzi w nim Raspberry Pi, płytka stykowa i szereg przewodów).

Produkty masowe - podbijamy światowe rynki

Zacznijmy od produktów masowych, które mają dość wyraźne cechy i wymagania. Załóżmy że masz gotowy układ kamery z monochromatyczną matrycą i filtrami do fotografii w ultrafiolecie i podczerwieni - coś czego zwykły aparat nie potrafi. Zrobiłeś wstępną analizę i wynika że taki produkt znajdzie wystarczającą ilość nabywców. Załóżmy że działa to tak: mini PC jak np. Raspberry Pi steruje małą kamerą przemysłową oraz poprzez GPIO prostym kołem filtrowym zmieniającym filtry. Do tego zasilanie z akumulatora, wyświetlacz i inne komponenty. **Jak z płytki stykowej przejść do produktu konsumenckiego?**

Proces przejścia od prototypu do gotowego produktu nie będzie ani krótki, ani tani. Aparat musi być relatywnie mały i trwały. Oznacza to że elektronika

powinna być zintegrowana na płytce PCB, tak by nie było konieczności stosowania różnego rodzaju przewodów i innych nietrwałych lub nieporęcznych konstrukcji. Tworzeniem projektów i wykonaniem płytek PCB zajmują się wyspecjalizowane fabryki, w których automaty potrafią budować tysiące jak nie setki tysięcy gotowych układów w krótkim okresie czasu. Twoim pierwszym krokiem powinno być nawiązanie współpracy z taką fabryką - w celu stworzenia i zoptymalizowania PCB dla układu, a następnie by rozpocząć produkcję. Jako etap pośredni można skorzystać z sieciowych usług takich jak 123d.circuits.io, gdzie możesz narysować swój układ, a następnie wygenerować projekt PCB i zamówić kilka sztuk, czy też poprosić społeczność o porady przy podstawowej optymalizacji płytki.

Prędzej, czy później dojdziesz do etapu składania zamówienia w fabryce na PCB. Problem w tym że takie fabryki przyjmują zamówienia na duże ilości by było to opłacalne. W zależności od skomplikowania układu mogą to być setki, a nawet tysiące sztuk. 5000 sztuk po 50 zł każda to już 250 tysięcy złotych. Bez zewnętrznego wsparcia finansowego mało których startup sprzętowy byłby w stanie wyłożyć takie pieniądze. Na szczęście na ratunek przychodzi **crowdfunding**, czyli zbiórki na Kickstarterze, Indiegogo i podobnych serwisach. Dzięki takim zbiórkom udało zrealizować się wiele projektów, które inaczej nie byłyby w stanie sfinansować budowy. Dodatkowa zaleta to weryfikacja produktu w oczach przyszłych konsumentów - jeżeli dobrze przeprowadzona zbiórka nie przekłada się na prognozowany poziom zainteresowania to może jednak Twój pomysł nie jest aż tak dobry?

Projekt Kano zebrał półtora miliona dolarów (zestaw edukacyjny złożony z Raspberry Pi, pomarańczowej klawiatury i materiałów edukacyjnych). Udoo, czyli połączenie Arduino z dość silnym mini PC zebrał 650 tysięcy, choć jako minimum wyznaczył sobie jedynie 27 tysięcy. Z drugiej strony są też projekty, które nie osiągnęły swoich kwot minimalnych. W chwili pisania tego artykułu na Kickstarterze trwa zbiórka na projekt SnaPiCam - czyli aparat cyfrowy na bazie Raspberry Pi i dedykowanej modułu kamery. Za 30 funtów (jakieś 150 zł) możemy dostać obudowę. Do tego musimy doliczyć koszt Raspberry, modułu kamery, wyświetla, akumulator i innych niezbędnych części. Nie prościej kupić gotową i lepszą cyfrówkę w sklepie? Dla wielu konsumentów tak zapewne jest i nic nie wskazuje by ten projekt osiągnąć dobry wynik (o ile dojdzie do minimum). Jeżeli uważasz jako programista i twórca że twój produkt jest dobry to **zainwestuj też w ludzi od marketingu i finansów**. Bez tego nie sprzedasz efektywnie produktu w crowdfundingu jak i możesz mieć problemy z wyceną całego przedsięwzięcia. Dobre *opakowanie* jest kluczowe.

Jeżeli tworzysz nowe urządzenie i chcesz **wprowadzić je do obrotu** to czeka Cię jeszcze spełnienie wymogów prawnych. Znaczek zgodności CE na urządzeniach jakie znajdziesz w sklepach nie bierze się sam z siebie. Musisz znaleźć regulacje prawne dopuszczenia do obrotu urządzeń z danej kategorii i upewnić się że Twoje urządzenie spełnia wszystkie wymagania. W przypadku kontroli urządzenia nie spełniające tych wymagań mogą marnie skończyć. Zainteresowanym polecam np. wyniki kontroli tabletów przeprowadzonej przez UKE. Wyobraź sobie sytuację,

w której musisz wymienić zasilacze dołączone do każdego tabletu, a tych na magazynie masz np. tysiące. Musisz więc uwzględnić **obsługę prawną** twojego przedsięwzięcia.

Niektóre projekty sprzętowe stosują swego rodzaju optymalizację **wykorzystując istniejące urządzenia**. Najłatwiej o to, gdy tworzysz nową usługę w oparciu o istniejące urządzenia. Np. do sieci klubów wprowadzasz karty lojalnościowe oparte o układ RFID. Potrzebujesz więc kart RFID jak i czytników i terminali. Można połączyć np. gotowy czytnik RFID i np. tablet by stworzyć działający terminal. Dzięki temu nie musisz wprowadzać produktu do obrotu, a jedynie tworzysz oprogramowanie, czy optymalizujesz dobór sprzętu. W przypadku działalności międzynarodowej możesz korzystać ze sprzętu dostępnego na danym lokalnym rynku. Odpada logistyka, czy cło i podatki przy przekraczaniu granicy UE. Tworząc dedykowane urządzenie można by zadbać o dodatkowe funkcjonalności, czy lepszy interfejs, ale nie zawsze jest to opłacalne, bo zyskać można niewiele, a wydać trzeba wtedy bardzo dużo. Przykładem takiego produktu może być Social WiFi, który wykorzystuje istniejące na rynku routery by dostarczyć nowe usługi, wcześniej nieobecne dla tego typu urządzeń.

Ręczna drukarka 3D **3D Simo** przeszła odwrotną drogę. Jest zupełnie nowym produktem i musiała przejść całą drogę jaką opisałem w poprzednich paragrafach. Dopiero po kilku zbiórkach crowdfundingowych, po wykonaniu szeregu prototypów twórcy mogli zacząć produkcję urządzenia i wysyłkę do osób, którzy zakupili je w zbiórkach - z kilkumiesięcznym opóźnieniem. Tutaj głównym problemem nie była elektronika, ale np. wykonanie form do bicia obudów, czy zamawianie części od różnych dostawców (np. dysz, czy układów grzewczych). Znaczek CE też pojawił się na obudowie drukarki. Odbiorca produktu tego nie widzi, ale dla twórców były to długie miesiące ciężkiej i czasami zapewne stresującej pracy. Z drugiej strony weszli na rynek z nowym produktem, a to już jest coś.

Produkty *unikatowe* - czyli radosna twórczość na małą skalę

Ręcznie tworzone urządzenia, czy usługi na nich oparte zdarzają się znacznie częściej niż projekty na masową skalę. Chcesz rozwiązać jakiś problem, dostarczyć jakieś proste urządzenie dla jednego klienta? Wtedy sprawa jest prostsza bo odpadają wysokie koszty, problemy prawne czy technologiczne. Wystarczy że stworzysz urządzenie i/lub oprogramowanie, upewnisz się że wytrzyma trudy pracy i gotowe. Twórcza i niecodzienna praca może być bardzo wciągająca.

Możesz zapytać **czy jest zapotrzebowanie na takie projekty?**. Otóż jest, choć może niezbyt duże. Wspomniałem już o terminalu z RFID. Jedna firma, z którą kiedyś współpracowałem używała technologii RFID na przyjęciach organizowanych z jednym z ich klientów. Niestety dość rzadko, bo firma zapewniające obsługę RFID wysoko się ceniła. Z drugiej strony wymagania stawiane RFID na tych imprezach były bardzo proste - po odczytaniu tagu sparować go z kontem Facebooka użytkownika (dać stronę logowania Facebooka...). Tablet, czy mini

PC z czytnikiem podłączonym na USB (np. udającym klawiaturę) dałoby się łatwo oprogramować by zapewnić takie funkcjonalności niskim kosztem.

Kolejny przykład to panel reklamowy, który miał za zadanie wyświetlać zdjęcia produktu po zeskanowaniu jego kodu kreskowego. Można to zrobić stosując prosty czytnik kodów kreskowych (który udając klawiaturę jest łatwy do oprogramowania), albo też tablet z kamerą i zewnętrznym ekranem na HDMI (gdzie wbudowana kamera wykrywa i skanuje kod kreskowy).

Dobór sprzętu W *amatorskich* projektach na niską skalę można poświęcić nieco czasu na dobór komponentów. Może to znacznie ułatwić stworzenie gotowego urządzenia, czy zredukować koszty. Jeżeli jesteś w stanie zrobić coś bez użycia mini PC tylko za pomocą samego mikrokontrolera to warto do tego się przyłożyć. Z drugiej strony mini PC mogą mieć przewagę jeżeli możesz skorzystać z gotowych urządzeń USB i systemowych bibliotek do ich obsługi. Krótszy czas poświęcony na napisanie oprogramowania też zmniejsza koszty.

Praktyczność łączy się z pomysłowością, innowacyjnością. Nie jestem w stanie podać na to wzoru, reguły. To już zależy od Ciebie i Twojego projektu. Mogę jedynie podać Ci kilka przykładów, czy pomysłów jakie możesz wykorzystać, czy też użyć jako punkt wyjścia.

Raspberry Pi może być mini komputerem o jakim pomyślisz w pierwszej kolejności gdy zajdzie potrzeba wykorzystania takowego. A co jeśli będziesz potrzebował czegoś mocniejszego, albo mniejszego. Są inne mini komputery z silniejszymi procesorami, ale są też droższe. A gdyby zamiast mini komputera dla miłośników elektroniki użyć mini komputera dostępnego powszechnie na rynku konsumenckim? Androidowe dongle HDMI są takim urządzeniem. Te wyposażone w znacznie wydajniejszy czterordzeniowy układ RK3188 z 2GB RAM nie dość że są już tańsze od Raspberry (ceny w chińskim sklepie dx.com), to oferują nam Androida, czy też zwykłego Linuksa (Picuntu, Linuxium), choć z nieco ograniczoną obsługą (brak obsługi VPU). W małej obudowie kryje się moduł WiFi, a na USB możemy podłączyć potrzebny nam czytnik, lub inne urządzenie. Wyjście HDMI możemy podłączyć od razu do ekranu. Nieco droższe i większe dongle zaoferują nam też złącze Ethernetowe.

Androidowe dongle, czy też tablety, albo taniejące nettopy z procesorami Intela, czy AMD mogą dostarczyć nam potrzebnej wydajności, ale nie zaoferują nam pinów GPIO. Wybranie odpowiednio silnego mini PC z pinami GPIO może okazać się problematycznym wyborem - np. mini PC Radxa Rock da nam ten sam układ RK3188 co dongle, ale przynajmniej na chwilę obecną obsługa GPIO tego mini komputera zostawia wiele do życzenia pod względem dokumentacji, bibliotek, czy przykładowego kodu. Co wtedy? Na ratunek mogą przyjść nam płytki z mikrokontrolerami. Możemy podłączyć je poprzez USB, czy w przypadku tabletów (lub smartphonów) bez odpowiedniej obsługi USB host i z brakiem sterowników do układów UART możemy skorzystać z komunikacji poprzez Bluetooth (gdzie mikrokontroler wykorzystać będzie układ do

komunikacji szeregowej poprzez Bluetooth).

Jeżeli jesteś w stanie przeboleć użycie C to otwiera się przed tobą świat Arduino - od dużych płytek do prototypowanie aż po małe płytki do urządzeń miniaturowych. Jeżeli potrzebujesz obsługi Ethernetu, WiFi, czy Bluetooth to też coś się znajdzie. Arduino oferuje SDK dla wielu różnych płytek (gdzie Raspberry trzyma się jednego urządzenia).

Rodzina mikrokontrolerów na platformie mbed wykorzystuje C++, choć nie jest to kluczową cechą. Największa zaleta to programowanie przez przeglądarkę za pomocą webowego IDE i zdalnego kompilatora, który przygotuje nam plik, jaki należy wrzucić na nasz mikrokontroler. Przydatne jest też duże zróżnicowanie płytek dostępnych na tej platformie. Za najbardziej rozbudowane (np. te z obsługą USB Host i Ethernetem) zapłacimy kilkaset złotych. Natomiast za najprostsze bez takich funkcjonalności około 50 - 60 zł.

MicroPython jest opcją dla chcących wykorzystać Pythona. Obecnie dostępna jest tylko płytka PyBoard za około 30 funtów. Jeżeli Chińczycy zaimplementują MicroPythona na jednej ze swoich płytek z kompatybilnym mikrokontrolerem to wtedy powinny pojawić się tańsze płytki. PyMCU jest tańsze, ale wymaga połączenia z komputerem by działać. Obie pythonowe platformy niestety nie oferują tylu bibliotek i przykładów co Arduino. Może to mieć kluczowe znaczenie, gdy będziesz musiał wykorzystać inne układy komunikujące się za pomocą SPI, czy I2C.

Czasami można zupełnie odejść od mikrokontrolerów i takiej elektroniki "dla twórców". Jeżeli potrzebujesz bardziej złożonego urządzenia, np. kamery przemysłowej, czy laboratoryjnej to nie zrobisz jej z prostych modułów dla Arduino. Gotowe kamery tego typu nie dostarczają na ogół bibliotek do Pythona, ale też muszą wystawiać jakieś API, żeby aplikacje mogły z nich korzystać. Pod MS Windows dość często jest to API dla .NET. Jeżeli tak się dzieje to możemy wykorzystać IronPythona do oskryptowania urządzenia. Sprzęt astronomiczny operuje na standardzie ASCOM, który dostarcza API nie tylko dla .NET, ale też wystawia obiekty COM, które można wykorzystać w zwykłym Pythonie. Dzięki temu możesz zyskać dostęp do zaawansowanych kamer, czy kół filtrowych. Jeżeli pod Linuksem twórca sprzętu daje binarny plik *.so z interfejsem do urządzenia to też da się go wykorzystać z poziomu Pythona za pomocą modułu ctypes.cdll.

Na zakończenie

Mam nadzieję że zainteresowałem Ciebie elektroniką i możliwościami jakie dają projekty sprzętowe - czy to dla własnej zabawy, czy dla nowych klientów Twojej firmy. Zaprezentowałem różne przykłady, różne platformy sprzętowe. Możesz użyć ich jako punkt wyjścia we własnych przygodach z elektroniką. Rynek rozwija się dość dynamicznie i za parę miesięcy nie wszystkie zawarte tu informacje mogą być już aktualne.

W sieci

- http://www.dx.com/c/electrical-tools-499/arduino-scm-supplies-436 - chiński sklep z ogromną ilością elektroniki do Arduino i przyjaciół.
- http://kamami.pl, http://nettigo.pl, http://botland.com.pl, http://electropark.pl, http://arduinosolutions.com - polskie sklepy z elektroniką i komponentami
- http://www.circuitsforfun.com - strona PyMCU
- http://micropython.org - strona MicroPythona i PyBoard
- http://www.python.rk.edu.pl/w/p/elektronika-i-python/ - moja strona poświęcona Pythonowi *w elektronice*

Journey to the center of the asynchronous world

Introduction

Since the release of Python 3.4, one of the hottest and most frequently mentioned topics is the asyncio module, introduced in PEP 3156. In the following short article I'll try to show you some cool stuff that lies at the core of this module. But before we dig in, there is one warning: most of the samples presented in this article are written in Python 3.4, so make sure to use at least that version when running the examples. These can be found at my github account.

Generators

Let's start with this sample piece of code, generator1.py:

```
def countdown(n):
    while n > 0:
        yield n
        n =- 1
```

As you've probably noticed, this is one of the simplest generator functions imaginable. Its typical usage is as following:

```
for x in countdown(10):
    print("Got ", x)
```

This prints every number starting from 10 down to 1. So we can conclude that every function written using the `yield` statement is a generator, which can then be used to feed all kinds of loops and iterations. If we look under the hood, this iteration calls `next()` to get the next value from the generator, until it reaches the `StopIteration` exception. We can illustrate that with the following piece of code:

```
c = countdown(3)
print(c)
next(c)
next(c)
next(c)
next(c)
```

The result of running this code is:

```
<generator object countdown at 0x7f2b0fa7a0d0>
3
2
1
Traceback (most recent call last):
  File "generator1.py", line 21, in <module>
    print(next(c))
StopIteration
Traceback (most recent call last):
  File "stdin", line 1, in ?
    print(next(c))
StopIteration
```

This is just the beginning, to make sure that everyone is on the same page. Expect more of the promised awesomeness to come! What is probably a lesser known fact, is that the `yield` statement can be used to receive values, generator2.py:

```
def receiver():
    while True:
        item = yield
        print("Got: ", item)

c = receiver()
print(c)
next(c)
c.send(43)
c.send([1, 2, 3])
c.send("Hello")
```

The output of the above code is as following:

```
<generator object receiver at 0x7f1690d88f58>
Got:  43
Got:  [1, 2, 3]
Got:  Hello
```

This was introduced in PEP 342, where the idea of coroutines appeared for the first time. This PEP extends the functionality of the generators presented in the first example with the possibility to send values to the generators. Essentially any function having the `yield` statement in its body is actually a generator. This means that it is not going to execute, but rather it will return a generator object, which provides the following operations: * `next()` - advance code to the `yield` statement and emit a value, if such was passed as a parameter. That's the only operation you can call after creating the generator. * `send()` - sends a value to the `yield` statement, making it produce a value instead of emitting one. Remember to call `next()` beforehand. * `close()` - a way to inform the generator that it should finish its work. It generates the `GeneratorExit` exception upon calling the `yield` statement. * `throw()` - gives you the opportunity to send an error to generator upon a call to the `yield` statement.

In Python 3.4 specifically you can have both the `yield` and the `return` statement. In previous Python versions this would be a syntax error. Currently if you write, generator3.py:

```
def returnyield(x):
    yield x
    return "Hi there"

ry = returnyield(5)
print(ry)
print(next(ry))
print(next(ry))
```

The output of the above code is as following:

```
<generator object returnyield at 0x7f27bf38bf58>
5
Traceback (most recent call last):
  File "generator3.py", line 15, in <module>
    print(next(ry))
StopIteration: Hi there
```

If you carefully study the output, you'll notice that the value of the `return` statement is actually passed as a value of the `StopIteration` exception. Interesting, isn't it?

Delegating to a subgenerator

PEP 380 proposed the syntax for a generator to delegate part, or all, of its work to another generator. This basically means that instead of manually iterating, we're passing the generation to somebody else, who will do it for us. This is presented in yieldfrom.py:

```
def yieldfrom(x, y):
    yield from x
    yield from y

x = [1, 2, 3]
y = [4, 5, 6]
for i in yieldfrom(x, y):
     print(i, end=' ')
```

The expected output is a series of numbers from 1 to 6. What is happening here is that both of these `yield from` statements take values from both lists, consume them and join them as if they were one list. In its simplest form, these can be seen as hidden `for` loops, but soon you'll see there's more to it. Beyond this, there is the concept of generator chaining, meaning iteration can be delegated even further. Let's create something more complicated:

```
for i in yieldfrom(yieldfrom(a, b), yieldfrom(b, a)):
    print(i, ' ')
```

In the above example, the outermost call will delegate iteration to the inner generators until we reach a single value that will be yielded.

Context managers

Let's leave the generator and coroutines topic for a bit and look at something different. I'm hoping the reader is familiar with these constructs:

```
file = open()
# do some stuff with file
file.close()

lock.acquire()
# do some stuff with lock
lock.release()
```

These constructs are currently nicely handled by context managers, introduced in PEP 343 - the `with` statement. Context managers are basically normal objects implementing two methods: * `__enter__(self)` - start work with your object, returning it * `__exit__(self, exc, val, tb)` - release the object, or handle the exception

Let's create a simple context manager for working with a temporary directory, contextmanager1.py:

```
class tempdir(object):
    def __enter__(self):
        self.dirname = tempfile.mkdtemp()
        return self.dirname
    def __exit__(self, exc, val, tb):
        shutil.rmtree(self.dirname)

with tempdir() as dirname:
    print(dirname, os.path.isdir(dirname))
```

This sample context manager will create a temporary directory, whose name will be printed and then checked for it's existence. Thanks to the awesome Python core developers, the `yield` statement and the `@contextmanager` decorator, the above code can be rewritten as follows, contextmanager2.py:

```
@contextmanager
def tempdir():
    dirname = tempfile.mkdtemp()
    try:
        yield dirname # the magic happens here
    finally:
        shutil.rmtree(dirname)
```

You will use this piece of code in exactly the same way as in the previous context manager. The only difference is how you define your context manager. In the latter example, the decorator is creating the context manager for you, and `yield` returns the temporary directory. If you look under the covers you'll see that calling `tempdir()` in the first example will return `<__main__.tempdir object at 0x7f3e4778f5a0>` whereas the later - `<contextlib._GeneratorContextManager object at 0x7fd94c7ce538>`. Do you see the difference? If you look closely at the `@contextmanger` decorator you'll find out that it sets up the `__enter__()` and `__exit__()` methods, with some additional error checking, see: contextlib.py#96. For those of you concerned about performance, my test shows the decorator solution runs ~9% slower than its class counterpart, but think of how much easier to read the decorator solution is.

Asynchronous processing

Finally we've reached the last part - asynchronous processing. The usual way of processing in such cases is: we have some main thread; in it we run some asynchronous function, and after some time we reach for the results. This is a very common programming pattern, which can be presented with the following code, future1.py:

```
def executor(x, y):
    time.sleep(10)
    return x + y

pool = ThreadPoolExecutor(8)
fut = pool.submit(executor, 2, 3)
fut.result()
```

The above code runs in a different thread, but here we're blocked; we wait until we get the result. The next example shows how to use callback functions that will return when the result is ready, whereas in the meantime we still have control over the main thread, future2.py:

```
def handle_result(result):
    """Handling result from previous function"""
    print("Got: ", result.result())

pool = ThreadPoolExecutor(8)
fut = pool.submit(executor, 2, 3)
fut.add_done_callback(handle_result)
```

A quick note: if an exception happens inside the executor method, it will be returned when getting the result. Testing this will be left as an exercise to the reader.

`asyncio` basics

OK, we've reached a point where I've shown you a couple of cool tricks with generators, but you may be asking "How is this useful? What can we do with it?" Let's then move to the final part where I'll show you how, using previous parts we can bypass certain Python limitations and create `asyncio` core functionality.

Let's start with creating a task object, which is similar to what I've shown you before, but this time, we'll put the idea into a reusable object, task1.py:

```
class Task:
    def __init__(self, gen):
        self._gen = gen

    def step(self, value=None):
        try:
            fut = self._gen.send(value)
            fut.add_done_callback(self._wakeup)
        except StopIteration as exc:
```

```
            pass

    def _wakeup(self, fut):
        result = fut.result()
        self.step(result)
```

If you look at the code you'll notice it is almost identical to the previous one, but placed inside of some sort of a context manager class. The only difference being method names, `step()` in place of `__enter__()` and `_wakeup()` for `__exit__()`. What we have here, actually, is a task object accepting a generator as the only initialization parameter, with the main function `step()` responsible for advancing the generator to the next `yield` statement, sending in a value and a callback to do something with the result. There's also one little trick at the end of the attached callback method called `_wakeup()` where we feed ourselves with the result to proceed execution to the next `yield` statement.

So let's create a recursive function, to show that using the above tricks we can bypass Python's recursion limit which by default is 1000.

```
def recursive(pool, n):
    yield pool.submit(time.sleep, 0.001)
    print(n)
    Task(recursive(pool, n+1)).step()
```

If you run it long enough, you'll notice that in using this little trick Python doesn't have a stack limit any more. What's more, the execution does not provide any overhead when run. You should definitely check it if you don't believe me.

There's still one more modification to our `Task` object which I'd like to show you. So far, this class can only process background tasks, but how do you return something from that background task? Let's use the `concurrent.futures.Future` object as a base class for our `Task`. To perform this we need to do a little Python patching, meaning we need to make the `Task` class iterable to be used inside the `yield from` statement:

```
def patch_future(cls):
    def __iter__(self):
        if not self.done():
            yield self
        return self.result()
    cls.__iter__ = __iter__
```

Then we can properly modify our `Task` object to return the result:

```
class Task(Future):                         # <--

    def __init__(self, gen):
        super().__init__()                  # <--
        self._gen = gen

    def step(self, value=None):
        try:
            ...
        except StopIteration as exc:
            self.set_result(exc.value)  # <--
```

Now we can use the `Task` object to do some intensive calculation and retrieve the result, task2.py.

```
def calc(x, y):
    print("I'm going to sleep for a while...")
    time.sleep(10)
    return x + y

def do_calc(pool, x, y):
    result = yield from pool.submit(calc, x, y)
    return result

if __name__ == '__main__':
    pool = ProcessPoolExecutor(8)
    t = Task(do_calc(pool, 2, 3))
    t.step()
    print("Got ", t.result())
```

Summary

The attentive reader might say at this point, "We're at the summary already, but you've promised to show what `asyncio` internals look like!" But I just did that: the last example of the `Task` object is almost exactly the same as the one in the `asyncio` module. For reference see tasks.py#25, with more error handling, and most importantly, some additional useful functions which hide implementation details from users to make it easier to use. Of course, there's also the most important thing, which is the event loop being the core runner instead of thread pools.

Hopefully, this article made you eager to get more from generators. I highly recommend reading David Beazley's trilogy on this topic:

1. Generator Tricks for Systems Programmers

2. A Curious Course on Coroutines and Concurrency
3. Generators: The Final Frontier

I also would like to thank him for giving me the chance to use his materials as an input for my article and presentation. Additionally, check out his awesome, mind-blowing book Python Cookbook.

Co tam, panie, w polityce? Czyli czym zaskoczył nas Python 3.4

Marcin Bardź

Python w wersji 3.4 światło dzienne ujrzał 16 marca 2014. To wydanie nie wprowadza żadnych zmian do samego języka, zamiast tego mamy kilka nowych bibliotek, usprawnienia w już istniejących modułach oraz wiele ulepszeń "pod maską".

Ta pozorna stagnacja w rozwoju języka została zaplanowana oraz wprowadzona w życie z pełną premedytacją (PEP 3003 o złowróżebnym tytule *Python Language Moratorium*) i ma ona na celu umożliwienie "dogonienia" bazowej implementacji (CPython) przez inne implementacje języka, takie jak Jython czy PyPy.

Brak zmian w składni nie oznacza jednak, że przeciętny pythonista nie dostanie do rąk nowych zabawek, ciekawych narzędzi, a jego życie nie stanie się jeszcze prostsze.

Nowe biblioteki

Najnowsza odsłona naszego ulubionego gada raczy nas pokaźną baterią całkiem nowych bibliotek, wśród których każdy użytkownik powinien znaleźć coś dla siebie.

`asyncio` Potężna biblioteka umożliwiająca tworzenie kodu współbieżnego, przełączany dostęp do zasobów we/wy, uruchamianie klientów/serwerów sieciowych, a to wszystko w jednym jedynym wątku! Dla osób znających Twisted nie będzie to nic nowego, mimo to wprowadzenie tak potężnego narzędzia do biblioteki standardowej otwiera wiele nowych możliwości.

Szczegółowy opis biblioteki wykracza daleko poza ramy niniejszego artykułu, dlatego wymienię tylko główne różnice pomiędzy Twisted i `asyncio`:

- `asyncio` jest prostsze i składa się z mniejszej liczby modułów. Z jednej strony `asyncio` nie ma wszystkich możliwości Twisted, z drugiej jednak

powinno być łatwiejsze w użyciu i nowy użytkownik powinien móc się szybciej wdrożyć (korzystniejsza krzywa uczenia się).
- Dokumentacja `asyncio` jest przejrzysta i ułożona w sposób intuicyjny dla przeciętnego programisty Pythona.
- `asyncio` wspiera najnowsze wersje Pythona i potrafi wykorzystać jego dobrodziejstwa (np. składnię `yield from`).

`ensurepip` Zaczęło się od PEP 453, który namaścił `pip` jako rekomendowane narzędzie zarządzania bibliotekami. Gdy już PEP został zaakceptowany, należało się więc upewnić, że użytkownik będzie miał dostęp do tego błogosławionego narzędzia.

Zasadniczo, standardowa instalacja Pythona powinna zawierać `pip` w wersji, która była najnowszą w momencie pojawenia się RC danego wydania. Jeśli jednak `pip` by się nam gdzieś zapodział (np. w wirtualnym środowisku), wystarczy wywołać z linii polecń:

```
$ python -m ensurepip
```

I już mamy pewność, że `pip` jest dostępny. Co ciekawe, żeby wykonać powyższą komendę, nie jest potrzebny dostęp do internetu, gdyż `ensurepip` przechowuje na swoje potrzeby lokalną kopię biblioteki `pip`.

Przeciętny użytkownik może nie musieć w ogóle igrać z modułem `ensurepip`, a cały ten cyrk wynika z faktu, że `pip` jest niezależnym projektem, posiadającym własny cykl wydawniczy.

`enum` Po wielu latach i po wielu niezależnych implementacjach, Python doczekał się w końcu swoich własnych typów wyliczeniowych. Dzięki nim można teraz pisać elegancki i mniej podatny na błędy kod.

W myśl zasady, że jedna linijka kodu znaczy więcej, niż tysiąc słów, przedstawiam poniżej próbkę możliwości modułu:

```
>>> from enum import Enum
>>> class Osoba(Enum):
...     ja = 1
...     ty = 2
...     on_ona_ono = 3
...     ona = 3
...     ono = 3
...
>>> Osoby = Enum('Osoby', 'my wy oni_one', module=__name__)
>>> być = {
...     Osoba.ja: "jestem",
```

```
...     Osoba.ty: "jesteś",
...     Osoba.on_ona_ono: "jest",
...     Osoby.my: "jesteśmy",
...     Osoby.wy: "jesteście",
...     Osoby.oni_one: "są"
... }
>>> for lp, lmn in zip(Osoba, Osoby):
...     poj = "{}. {} {}".format(lp.value, lp.name, być[lp])
...     mn = "{}. {} {}".format(lmn.value, lmn.name, być[lmn])
...     print("{:20}{:20}".format(poj, mn))
...
1. ja jestem        1. my jesteśmy
2. ty jesteś        2. wy jesteście
3. on_ona_ono jest  3. oni_one są
>>> print("Myślę, więc %s" % być[Osoba.ja])
Myślę, więc jestem
>>> print(Osoba(3))
Osoba.on_ona_ono
>>> print(Osoba.ono)
Osoba.on_ona_ono
>>> Osoby['oni_one']
<Osoby.oni_one: 3>
```

`pathlib` Kolejna obszerna biblioteka, przenosząca operacje na ścieżkach i plikach z prehistorii do świata programowania obiektowego. Moduł łączy w sobie funkcjonalności `os.path`, `glob` oraz wielu funkcji z innych bibliotek (głównie `os`), opakowując wszystko w przepyszną obiektową otoczkę.

Klasy biblioteki podzielone są na dwie odrębne części:

- Czyste ścieżki (*pure paths*) - umożliwiające operacje na samych ścieżkach.
- Konkretne ścieżki (*concrete paths*) - dodające jeszcze operacje we/wy.

Ponadto, moduł udostępnia dwa warianty klas, jeden dla systemów POSIXowych i jeden dla ścieżek Windowsowych, ale zwykły zjadacz chleba do szczęścia potrzebuje tylko jednej klasy `Path`, która reprezentuje konkretną ścieżkę zgodną z aktualnym systemem operacyjnym.

Czasy poszukiwań potrzebnych funkcji plikowych odchodzą w niepamięć, gdyż użycie `pathlib` jest tak wygodne i intuicyjne, że aż trudno uwierzyć, że to cudeńko pojawiło się w bibliotece standardowej dopiero teraz.

Oto kilka przykładów użycia `pathlib`:

```
>>> from pathlib import Path
>>> p = Path('/')
```

```
>>> p.is_dir()
True
>>> q = p / 'etc' / 'resolv.conf'
>>> q
PosixPath('/etc/resolv.conf')
>>> q.exists()
True
>>> q.owner()
'root'
>>> with q.open() as f:
...     print(f.readline())
...
#

>>> q.parts
('/', 'etc', 'resolv.conf')
>>> q.parents[1]
PosixPath('/')
>>> q.parents[2]
Traceback (most recent call last):
...
raise IndexError(idx)
IndexError: 2
>>> q.name
'resolv.conf'
>>> q.suffix
'.conf'
>>> q.relative_to('/home')
Traceback (most recent call last):
...
ValueError: '/etc/resolv.conf' does not start with '/home'
>>> list(q.parent.glob('a*.conf'))
[PosixPath('/etc/asl.conf'), PosixPath('/etc/autofs.conf')]
```

`selectors` Moduł ten udostępnia wysokopoziomowe mechanizmy przełączania we/wy. Jest to abstrakcyjny i w pełni obiektowy, a co za tym idzie łatwiejszy w użyciu odpowiednik niskopoziomowej biblioteki `select`.

Trik z parą modułów służących do realizacji tego samego zadania nie jest niczym nowym w Pythonie, zastosowano go już wcześniej do obsługi wielowątkowości (niskopoziomowy `_thread` i wysokopoziomowy `threading`).

Programista otrzymuje do ręki (między innymi) klasę `DefaultSelector`, która jest abstrakcją najbardziej efektywnej implementacji selektora dla danej platformy, a dzięki wspólnej klasie bazowej `BaseSelector`, dostępny jest jednolity interfejs obsługi, co przekłada się na czytelny, niezależny od platformy kod, bez

niepotrzebnych klauzul `if`.

`statistics` Tym modułem twórcy Pythona starają się uszczęśliwić statystyków, księgowych, maklerów oraz wszystkich pozostałych im podobnych. Znajdziemy tu funkcje liczące średnią, medianę, odchylenie standardowe i kilka innych tajemniczych rzeczy pokroju wariancji.

Wśród wszystkich tych funkcji, na szczególną uwagę zasługuje `mode()`, wyszukująca najczęściej występujący element w dyskretnym zbiorze danych (lub rzucająca wyjątkiem `StatisticsError`, jeśli takiego elementu nie ma). Funkcja `mode()` bywa przydatna nie tylko księgowym i może czasem zaoszczędzić kilka linijek kodu.

Moduł zapewne będzie się jeszcze rozwijał, obrastając w nowe funkcje, gdyż na chwilę obecną zawiera tylko niewielki wycinek tego, co dla dobra ludzkości wymyślili statystycy.

`tracemalloc` Ostatnia na liście, ale zdecydowanie warta uwagi biblioteka, która pozwala na tworzenie migawek (*snapshot*) alokowanych bloków pamięci oraz przetwarzanie ich na różne sposoby.

Moduł udostępnia trzy rodzaje informacji:

- Traceback miejsca alokacji obiektu.
- Statystyki przydzielonej pamięci (dla pliku, dla linii kodu).
- Porównywanie migawek w celu wykrycia wycieków.

Poniższy plik (nazwałem go `t.py`) pokazuje przykładowe użycie niektórych funkcji biblioteki:

```
01  import tracemalloc
02
03  tracemalloc.start()
04
05  a = 'A'*1000000
06  b = 'ź'*1000000
07  c = list(range(1000000))
08  d = set(range(1000000))
09  e = {_: None for _ in range(1000000)}
10
11  snap = tracemalloc.take_snapshot()
12  top = snap.statistics('lineno')
13  for stat in top[:5]:
14      print(stat)
15
```

```
16  f = list(range(1000000))
17  g = c + f
18
19  snap2 = tracemalloc.take_snapshot()
20  top2 = snap2.compare_to(snap, 'lineno')
21  print()
22  for stat in top2[:2]:
23      print(stat)
```

A tak wygląda efekt uruchomienia powyższego pliku:

```
$ python3.4 t.py
t.py:9: size=74.7 MiB, count=999744, average=78 B
t.py:8: size=58.7 MiB, count=999745, average=62 B
t.py:7: size=35.3 MiB, count=999746, average=37 B
t.py:6: size=1953 KiB, count=1, average=1953 KiB
t.py:5: size=977 KiB, count=1, average=977 KiB

t.py:16: size=35.3 MiB (+35.3 MiB), count=999745 (+999745), average=37 B
t.py:17: size=15.3 MiB (+15.3 MiB), count=1 (+1), average=15.3 MiB
```

Wszystko jest widoczne jak na dłoni, litera `ź` zajmuje więcej miejsca niż `A`, jednak to wszystko nic w porównaniu z rozmiarem słownika. Tego rodzaju dane mogą być nieocenione przy debugowaniu oraz podczas optymalizacji kodu.

Na koniec muszę ostrzec, że `tracemalloc` dość mocno spowalnia wykonywanie programu, więc po pierwsze – żeby otrzymać wyniki czasem trzeba uzbroić się w cierpliwość, a po drugie – nie należy stosować `tracemalloc` w środowisku produkcyjnym.

Inne, co ciekawsze zmiany

Tryb izolowany Pythona można teraz uruchomić z parametrem `-I`, który odcina interpreter od `site-packages`, jak i od wszelkiego dostępu do zewnętrznych bibliotek. Użycie tego trybu jest zalecane przy zastosowaniu Pythona w skryptach systemowych.

Nowy format `pickle` i `marshal` Python 3.4 wprowadza nowe formaty dla `pickle` i `marshal` (odpowiednio 4 i 3).

W przypadku `pickle` rozwiązanych zostało wiele problemów występujących w poprzednich wersjach protokołu, a także poprawiona została wydajność. Tradycyjnie już, w celu zapewnienia wstecznej kompatybilności, nowy format `pickle` nie jest domyślnym, tak więc żeby go użyć, należy jawnie określić wersję protokołu, najlepiej (też tradycyjnie już) używając stałej `pickle.HIGHEST_PROTOCOL`.

Jeśli chodzi o `marshal`, to dzięki uniknięciu powielania niektórych obiektów, zmniejszył się rozmiar plików `.pyc` (i `.pyo`), a co za tym idzie, spadła ilość pamięci zajmowanej przez moduły wczytane z tychże plików.

***Single-dispatch* w `functools`** W module `functools` pojawił się niepozorny dekorator `singledispatch()`, który pozwala na zdefiniowanie funkcji generycznej, posiadającej różną implementację w zależności od typu argumentu.

Bez wdawania się w dywagacje, poniższy przykład powinien rzucić nieco światła na to, ile dobrego kryje się za tą mętną definicją:

```
>>> from collections.abc import Sequence
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg):
...     print("Łapię całą resztę, tym razem był to", type(arg))
...
>>> @fun.register(int)
... @fun.register(float)
... def _(arg):
...     print("Mamy numerek!")
...
>>> @fun.register(Sequence)
... def _(arg):
...     print("Sekwencja o długości ", len(arg))
...
>>> @fun.register(tuple)
... def _(arg):
...     print("Nie lubię tupli!")
...
>>> fun(11)
Mamy numerek!
>>> fun("tekścik")
Sekwencja o długości  7
>>> fun([1, 3, 5])
Sekwencja o długości  3
>>> fun((3, 4))
Nie lubię tupli!
>>> fun(set())
Łapię całą resztę, tym razem był to <class 'set'>
```

Poprawa bezpieczeństwa Nowy, bezpieczny, algorytm hashowania, obsługa TLS v1.1 i v1.2, możliwość pobierania certyfikatów z Windows system cert store, obsługa serwerowego SNI (Server Name Indication), bezpieczniejsze deskryp-

tory plików. To tylko niektóre z licznych zmian, wpływających korzystnie na bezpieczeństwo nowego Pythona.

Podsumowanie

Python 3.4 zdaje się być bardzo dobrym wydaniem, pomimo braku (a może dzięki brakowi?) zmian w składni. Wprowadza on szereg usprawnień oraz oddaje w ręce użytkownika kilka bardzo dobrych bibliotek wbudowanych, które z pewnością znajdą sobie miejsce w wielu plikach z rozszerzeniem `.py`.

Zarówno nowi użytkownicy, jak i starzy wyjadacze znajdą coś dla siebie w tym wydaniu, a dzięki praktycznie pełnej kompatybilności wstecz, nowy Python nie powinien też nastręczać problemów przy aktualizacji*.

* Oczywiście chodzi o aktualizację z Pythona 3.3 ;)

Źródła

- https://docs.python.org/3/whatsnew/3.4.html - oficjalny dokument "what's new" dla Pythona 3.4.
- https://mail.python.org/pipermail/python-ideas/ - lista mailingowa wyznaczająca nowe kierunki rozwoju Pythona.
- http://hg.python.org/cpython/ - repozytorium zawierające źródła (oraz nader ciekawe commit-logi) Pythona.
- http://legacy.python.org/dev/peps/ - propozycje usprawnień Pythona.

Proper code reviews

Tomasz Maćkowiak

Code quality is the ultimate goal that every programmer (and his manager) strives for. High code quality means fewer bugs, better performance and easier maintenance. One of the most popular ways of ensuring quality of the code (apart from peer programming) is to implement code reviews.

Not all code reviews are equal. Some developers are better at it, while others add no value in their comments.This article presents my approach to code reviews that has worked well for me and my team for about a year already.

What are code reviews?

This article is about informal, individual code reviews. We develop our code using Git version control system. The model we use is feature-branching, which

means that there is a new Git branch created off the master branch for each feature or bug fix. Developers work on this branch and, when done, they release a pull request. The pull request is then reviewed by at least 2 other developers. Either the original developer needs to address the review comments and release fixes for issues found during code review, or the pull request can be merged straight away.

Why do you do code reviews?

The most obvious answer to the above question is to keep high code quality. Two additional developers scrutinising all the code which gets into the repository are supposed to find mistakes, ineffective bits, convention violations and other issues that might be improved. The value of code review seems to be indisputable, but there are different ways of approaching reviews, some more beneficial than others.

Code quality is not the only benefit of code reviews. What are the others?

Standards

Thanks to code reviews, the team is bound to develop coding standards and adhere to them. Be it standards like PEP8, the usage of `%` operator against `format` function, to certain ways of using external libraries; over time the team reaches a consensus and everybody needs to act on its terms, unless they want to have their pull request rejected. At first there might be a few ways of doing something, but in the process of performing regular code reviews alternatives are being flagged and one winning solution emerges. In our team history we organised meetings following some particularly heated code review discussions, where we had to pick one standard over the other.

Knowledge sharing

If code reviews are performed properly (the whole team is obliged to carry them out and is notified about all pull requests and comments), then code reviews also serve the purpose of sharing knowledge among team members. That is how the above mentioned standards are being announced - team members learn from pull review's comments that a certain solution is preferable to the other. Also, the team learns other good practices. We are supposed to learn from our own mistakes, but learning from the mistakes of our colleagues is even better. The team starts to single out undesirable patterns if they are being flagged in code reviews. Provided that code review is done properly and the comments hold value (for example: *for performance use* `set` *here*), it can constitute an invaluable lesson. Another important function of code reviews is that they enable team members to track what is going on in the project. The team sees the code

being committed to the repository and they know what functionalities are being introduced and how they are implemented.

Ownership

The code is not supposed to be the private property of the developer who first wrote it. When practising code reviews, the reviewers share the responsibility for the code. If the code works poorly or does not work at all, the reviewers are guilty of letting its defects slip through their fingers. Another benefit is that there is always an opportunity for somebody else to fix the mistake in the code, because they are at least vaguely familiar with the code, since they browsed through it during code review.

Personal development

There is no faster way of improving your skill than to have somebody systematically point out your mistakes and (maybe) show you how to fix them. You improve your code every time you receive a comment in pull request. You gain knowledge and learn good practices from other people's pull requests. You post your own comments to other people's code and you are able to see if your opinions are valid.

How to do code reviews?

Code reviews only hold value if they are done properly.

Tools

One of the key factors in effective code reviews are tools. If submitting your code for pull request or reviewing the code takes too much effort, people will simply refuse to do it. The process needs to be automatised as much as possible. Email notifications about pull requests and comments help a lot, because even if you do not actively follow open pull requests, you are still notified about the progress. My recommendation is having a private Github. It supports email notifications, integrates into your source code repository, has a great interface and makes it really effortless to create a pull request as well as to review one.

The actual review

Depending on the size of the pull request, code review can take anywhere from 5 minutes to half a day in extreme cases. Since concentrating on the code under review requires a clear head, you need to allocate an appropriate amount of time

for the review. It is preferable to do review in-between tasks,at the beginning of the day (as a warm-up), or at the end of the day when you are too tired to keep on developing your own task anymore.

The code you are reviewing is most probably connected with a story or a bug in your issue tracking system. It is a good idea to first read about the issue so that you know what is to be implemented with the code.

In my opinion, you should never run the code you are reviewing. Running the code might give you a false impression that the code is acceptable and that it works as intended, while under the hood it is a mess.

Good review and bad review

What should you point out during code review to make your colleagues regard it as valuable input and not nit-picking?

Code review is done by qualified humans for a reason. You should not treat code review as another round of PEP8 checks - these should be done by the code's author before he creates a pull request. What if the author did not run his changes through PEP8 check? You still should not bother since doing something that a machine can do in under 1 second is never worth your time and effort. It is helpful to have PEP8 as part of your Continuous Integration suite as it prevents any potential PEP8 violation from slipping through to the master branch. Nobody appreciates comments about different possibilities of breaking newline in their code.

There are a few other types of comments that are annoying to developers, whereas, contrary to popular belief, they do bring in value. People very much dislike comments (for Python2) about changing strings to unicodes, for example in labels, especially if the string does not contain any non-ascii characters. Their argument is that *it doesn't matter anyway*, but in reality this error shows that the programmer does not understand the difference between byte-strings and text-strings. Even though such comments might seem annoying and repetitive, they do improve the code and foster the understanding of the language's fundamental structures.

A different kind of apparent nuisance is when the reviewer points out the inconsistencies of your code with the project's standards. For example, the usage of `format` might be preferred over the `%` operator. The usage of the latter is not an error per se but it goes against project policy. This kind of comment might be annoying, but in the long run abiding to the policy improves the coherence of the code and makes it easy to get into it by junior developers.

Another annoyance might be calling for docstring of functions or classes. Docstrings are almost always a good idea, unless you have a simple getter function with a sensible name; then it might not be needed. Asking for docstrings to

all functions in review might be alleviated by using automatic tools like pylint, which can do that for you.

During code review one might encounter pieces of code that are complex and non-trivial to understand at first glance, or require significant effort to analyze and understand. It is worth to ask the original developer to insert a comment describing the particularly hard bit so that programmers down the line will not have to spend half an hour analyzing the code just to learn what it was supposed to do.

When developers write comments and docstring, they usually need to use English which is a foreign language to most. The level of familiarity with the language is often visible in the comments. Incorrect grammar, misspellings etc. can make the comment hard to understand. It is a good idea to point out such issues during code review so that all your documentation is grammatically correct and understandable.

The most valuable comments during code review, though, are about the more high-level issues. Comments about architecture should be greatly appreciated. Pointing out that some code can be extracted to a helper or that some piece of code does not belong here but somewhere else, can greatly improve the maintainability of the code.

Quite often, experienced developers can point out that code under review does not need to be written at all, because there is already a built-in or a helper somewhere in the project that performs the same function. Such comments are usually only to be expected from highly experienced developers, but they carry a great learning potential for the more junior ones.

A similar case can be made with applying correct data structures and using efficient algorithms. An experienced developer can spot fragments of the reviewed code where the usage of a dictionary data structure can greatly improve its performance. Such a remark requires a higher order understanding of the purpose of the code - it goes beyond the meaning of a single line of the code, but pertains to the the overall result that the developer is trying to achieve.

It is fairly easy to review a code change where somebody has just added a few lines to a function. It is, however, much harder to analyze code that introduces conditional logic. This is usually the point at which the reviewer needs to stop and look deeply into what is happening. Analyzing the conditions and their effects can provide one of the most valuable feedbacks possible. The reviewer needs to ask himself the following questions: *what does this condition really mean?*, *when is this condition not met?*, *what happens when the condition is not met?*, *is this condition really needed?*. Some programmers try to solve problems by introducing a number of `if` statements with complicated conditions. In many instances such pieces of code can be refactored to a much simpler form, decreasing the code complexity significantly and making it much more readable.

An experienced reviewer is often able to provide (just by looking at the code) instances of corner cases in which the code would fail or predict a situation that

the original developer did not anticipate. Spotting such cases directly prevents live system bugs and is therefore of immense value. It often requires applying a very deep and meticulous review process, but the time spent on it is not wasted. You should not, therefore, cut down the amount of time spent on a pull request, but rather review it until you understand it fully.

Summary

Code review is one of the most valuable tools for assuring software quality. When done by experienced developers who devote enough time to it, it provides an opportunity to spot potential bugs, performance bottlenecks, or hard-to-maintain bits. Apart from the reviewers doing their job well, the developers under review also need to learn to embrace the critical comments and understand that code review is not a nuisance, but something that is done for their benefit.

Bayesian A/B testing with Python

Introduction

A/B testing is a simple yet powerful instrument to evaluate design decisions for web applications. It's a kind of behavioral research study akin to clinical trials, conducted to assess the effectiveness of alternative variations of graphical designs, wording solutions, logical decisions in terms of different performance metrics. It is to be contrasted to *personalization*: the goal of A/B testing is to find a solution that fits best to all users, whereas personalization aims to find a best solution for every particular user.

Traditionally, a classical framework of statistical hypothesis testing is used to evaluate the performance of variations. This text discusses the Bayesian approach to A/B testing, its pros and cons, and a way of implementing it using Python's `scikit` and an author's library called `trials`.

Classical approach

Suppose there are two different variations of the landing web page design: call them A and B. A common problem is to find the one that will probably produce more sign-ups based on collected data.

Within the classical frequentist approach, the following model is commonly used. Let A, B be two independent finite binary populations corresponding to all possible page views by target audience. Each item in population is either a 1 (success, user viewed the page and signed up) or 0, (failure, user viewed the page and didn't sign up). We assume that both populations are Bernoulli-distributed

with *fixed parameters* p_A, p_B. We don't know the parameters, but want to estimate them. This is an important point that will be returned to later: the parameters are *fixed yet unknown*.

We conduct an experiment by randomly showing the users different designs and logging the results. Sample data $X_A = \{x_A^{(i)}\}$, and $X_B = \{x_B^{(i)}\}$ are obtained, where $x_*^{(i)} \in \{0, 1\}$. We assume that every observation $x_A^{(i)}$, $x_B^{(i)}$ is randomly picked from the respective population in such a way that every item in population has an equal chance to be picked, i.e. X_A and X_B are randomly sampled from respective populations. Therefore, $x_A^{(i)} \sim \text{Bern}(p_A)$, $x_B^{(i)} \sim \text{Bern}(p_B)$. Moreover, this means that all observations $x_A^{(i)}$, $x_B^{(i)}$ are independent. This is a strong assumption that almost never holds in real life. It is natural that there exist some common factors that influence multiple observations (for example, geographic location, time zone, screen size, etc.). If the influence of these factors is quite substantial, then probably this model isn't best fit for the task.

One way to reason about the true population parameters p_A and p_B is to use two-sample t-test. We will test the null hypothesis about the equality of expected values of A and B:

$$H_0 : p_A = p_B \;\Rightarrow\; p_A - p_B = 0$$

Since p_A, p_B can be estimated by sample means $\hat{p}_A = \bar{X}_A$, $\hat{p}_B = \bar{X}_B$, the distribution of the test statistic $T = \frac{\bar{X}_A - \bar{X}_B}{\sigma_{\hat{p}_A - \hat{p}_B}}$ (standardized $\hat{p}_A - \hat{p}_B$) approaches standard normal as the number of observations gets large, and approximately equals the Student's t-distribution if the number is not large enough. It is straightforward to find the value of the test statistic, knowing a formula for estimated variance:

$$\sigma^2_{\hat{p}_A - \hat{p}_B} = \frac{\sigma^2_{X_A}}{n_{X_A}} + \frac{\sigma^2_{X_B}}{n_{X_B}}$$

And for the number of degrees of freedom ν of the t-distribution:

$$\nu = \frac{(\sigma^2_{X_A}/n_{X_A} + \sigma^2_{X_B}/n_{X_B})^2}{(\sigma^2_{X_A}/n_{X_A})^2/(n_{X_A} - 1) + (\sigma^2_{X_B}/n_{X_B})^2/(n_{X_B} - 1)}.$$

(Welch–Satterthwaite formula, for the the most general case).

Then we find corresponding data likelihoods under the null hypothesis $P(X \mid H_0)$. An α-confidence interval for the test statistic can be obtained as well: $\{T \pm \sigma_{\hat{p}_A - \hat{p}_B} \cdot t_{\nu,\alpha}\}$.

In practice, if the number of observations is big enough, z-test is often conducted instead, i.e. the T-statistic is assumed to be normally distributed.

To control the false positive rate we use α (usually 5%), which actually equals to the probability of a false positive $\text{P}(|T| < t_{\nu,\alpha} \mid \neg H_0) = \alpha$. The probability of a false negative $\text{P}(|T| \geq t_{\nu,\alpha} \mid H_0)$ is a called a power function of the test. For t-test, it depends on difference between p_A and p_B and the number of

observations. The larger is the number of observations and the larger is the difference between parameters, the smaller false negative rate is. For example, if the relative difference between true population parameters is small, e.g., 1%, around 80,000 observations is needed to provide at least 20% false negative rate. If the difference is 50%, 1000 is enough.

Problems

Frequentist treats a probability as a frequency of an event's occurrence in a number of repeated experiments. Frequentist techniques rely greatly on the Law of Large Numbers and Central Limit Theorem. For A/B testing we would want to get the results as quickly as possible, possibly before the CLT can be applied.

There's also a philosophical issue with hypothesis testing: it answers the wrong question. For the example above, what we would like to do is estimate the true population parameters based on the observed data. In other words, we want to know something like $\mathrm{P}(p_A > p_B \mid X)$. That formula doesn't make much sense within the frequentist paradigm, since probability is a frequency of an event in series of repeated experiments, yet p_A and p_B are *fixed* values, they're not outcomes of any experiments, not random variables, therefore, you can't make any probabilistic statements about them. The only thing hypothesis testing can provide is the likelihood of the data given the hypothesis, $\mathrm{P}(X \mid p_A > p_B)$. We need to pick a hypothesis, then check how well the obtained data supports it. That makes sense, but the inverse $\mathrm{P}(H \mid X)$ would fit our question so much better. In fact, the two probabilities $\mathrm{P}(X \mid H)$ and $\mathrm{P}(H \mid X)$ are connected by the Bayes rule, but one first has to change the definition of probability to apply it.

From a Bayesian perspective, probability is nothing but a degree of belief on a scale from 0 to 1. This interpretation not only drops the requirement for large repeated experiments, but allows to answer the question directly: what are our best estimates on population parameters given what we can observe.

Bayesian approach

Whereas p_A, p_B where fixed population parameters in the previous model, for Bayesian approach let p_A, p_B be independent random variables. The data now should be considered fixed. Previously, it was vice versa: parameters were fixed, and data was random; now parameters are random variables, and data is fixed.

Let *prior* distributions of p_A and p_B be Beta-distributed:

$$p_A \sim \mathrm{Beta}(\alpha_A, \beta_A),\ p_B \sim \mathrm{Beta}(\alpha_B, \beta_B)$$

The choice of Beta distribution will be explained later. Let number of sign-ups $k_A = |\{x_A^{(i)} = 1\}|$, number of page views $n_A = |X_A|$. Assume that the likelihood

of data obtained by logging views and sign-ups is binomial: $\mathrm{P}(X_A \mid p_A) = \mathrm{Binomial}(k_A; n_A, p_A)$. Analogically, for B.

Applying Bayes theorem, we can find the posterior:

$$\begin{aligned}
\mathrm{P}(p_A \mid X_A) &= \frac{\mathrm{P}(X_A \mid p_A) \cdot \mathrm{P}(p_A)}{\mathrm{P}(X_A)} \propto \mathrm{P}(X_A \mid p_A) \cdot \mathrm{P}(p_A) \\
&= \binom{n_A}{k_A} p_A^{k_A} (1 - p_A)^{n_A - k_A} \; \frac{1}{B(\alpha_A, \beta_A) p_A^{1-\alpha_A} (1 - p_A)^{1-\beta_A}} \\
&= \mathrm{Beta}(\alpha_A + k_A, \beta_A + n_A - k_A)
\end{aligned}$$

The nice result in the final step occurs because Beta is a conjugate prior to Bernoulli and Binomial distributions, and this is the reason why it was chosen as a prior for p_A and p_B.

From the posterior distribution we can find interesting values like $P(p_A > p_B \mid X)$, $P(p_A < p_B \mid X)$, and e.g., *lifts* $\frac{p_B - p_A}{p_A}$ and $\frac{p_A - p_B}{p_B}$, or pretty much any function we'd like using Monte Carlo techniques.

This model relies on similar assumptions (independent Bernoulli trials, implied by Binomial likelihood), but doesn't rely on large numbers like the previous one. An important observation is that instead of finding P(data | hypothesis) for a set of predefined hypotheses about the parameters, we integrate over all possible values of parameters (hypotheses) and get P(parameter | data) using Bayes theorem. In such sense, this approach is a generalization of hypothesis testing.

In the end, this approach doesn't suffer from the problems outlined previously. This comes at the cost of expensive MCMC computations for more complicated posteriors.

Example

Suppose we've gathered some data. Say, variation A was viewed 44 times, and produced 2 sign-ups, and variation B was viewed 96 times and produced 11 sign-ups. Let's use `scipy` to calculate some statistics.

```
from scipy import stats

data = {
    'A': { 'views': 42, 'signups': 2 },
    'B': { 'views': 85, 'signups': 11 }
}

posteriors = {
```

```
    variation: stats.beta(logs['signups'],
        logs['views'] - logs['signups'])
    for variation, logs in data.items()
}
```

Calculate expected sign-up rate $\mathrm{E}[p_A \mid X]$:

```
posteriors['A'].mean()
```

Get $ E[p_A|X] = 5.81%,E[p_B|X] = 13.37% $.

Calculate 95%-credible intervals:

```
lower = posteriors['A'].ppf(0.025)
upper = posteriors['A'].ppf(0.975)
```

$ P(1.00% < p_A < 14.41%) = 0.95 ,P(7.07% < p_B < 21.28%) = 0.95 $

Monte Carlo approach to compute $P(p_B > p_A \mid X) \approx \frac{1}{n} \sum_i \mathbb{I}[y_A^i > y_B^i]$ and expected lift:

```
import numpy as np

sample_size = 10000
samples = {
    variation: posterior.sample(sample_size) \
    for variation, posterior in posteriors.items()
}

dominance = np.mean(samples['B'] > samples['A'])
lift = np.mean((samples['B'] - samples['A']) \
    / samples['A'])
```

Variation B performs better, so $P(p_B > p_A) = 92.90\%$. Expected lift of sign-up rate under variation B is $+271.68\%$.

`trials` is a tiny library that does all of the above and a little bit more.

```
from trials import Trials

test = Trials(['A', 'B'], vtype='bernoulli')

test.update({
    'A': (2, 40),
    'B': (11, 79),
})
```

Statistics supported by `trials` for Bernoulli experiments: expected posterior, posterior CI, expected lift, lift CI, empirical lift, dominance.

$\mathrm{P}(p_A > p_B \mid X)$:

```
dominances = test.evaluate('dominance')
```

Expected lift $\mathrm{E}(\frac{p_B - p_A}{p_A} \mid X)$:

```
lifts = test.evaluate('expected lift')
```

Lift 95%-credible interval:

```
intervals = test.evaluate('lift CI', level=95)
```

Conclusion

We showed how to do A/B testing the Bayesian way using sign-up rate as an example metric. The technique is flexible enough to use any kind of interesting metrics by changing the prior and posterior distributions appropriately. Log-normal posterior could be used to evaluate variations based on time users spent on a page, Poisson posterior for number of clicks users made, etc. One could choose "nice" conjugate priors and posteriors, but, essentially, any distributions can be used at the cost of MCMC computation. The Bayesian approach is more general than the usual hypothesis testing approach, doesn't rely on large numbers in theory, and produces results that are easier to interpret.

Python w pracy vulnerability researchera/reversera

Wprowadzenie

Prezentacja ma na celu przedstawienie wykorzystania języka programowania Python, jako jednego z narzędzi wspomagających pracę osoby zajmującej się analizą oprogramowania pod kątem odnajdywania błędów bezpieczeństwa w dostarczonym oprogramowaniu. Przez analizę oprogramowania mamy na myśli w większości przypadków podejście blackbox, czyli źródła aplikacji/systemu nie są dostępne i to warsztat pracy, doświadczenie oraz stworzone narzędzia umożliwiają odnajdywanie nieznanych dotychczas błędów.

Słownik

- Vulnerability Researcher – Osoba zajmująca się analizą oprogramowania pod kątem odnajdywania błędów bezpieczeństwa.
- Reverser – Osoba zajmująca się analizą rewersyjną/wsteczną (Reverse Engineering).
- Vulnerability Research – Analiza oprogramowania pod kątem wykrywania błędów bezpieczeństwa.
- Vulnerability (vuln) – Podatność, błąd bezpieczeństwa. Przykładowo: DoS, LCE, RCE.
- DoS – Denial of Service
- LCE – Local Code Execution
- RCE – Remote Code Execution
- Reverse Engineering – Proces analizy kodu bez posiadania źródeł. Przykładowo mamy plik wynikowy EXE i dokonujemy analizy (statycznej bądź dynamicznej), w celu odtworzenia logiki aplikacji/algorytmów w niej użytych. Obszary zastosowania: Malware, Vulnerability Research, Cracking, Software Protection, DRM etc.
- Exploit - Konkretne wykorzystanie danej podatności. Click & Run.
- 0day - Nieznany szerszej grupie osób exploit. Pierwszy exploit wykorzystujący daną podatność.
- Fuzzing – Automatyczny proces testowania oprogramowania z wykorzystaniem poprawnych oraz niepoprawnych danych wejściowych (input).
- Input – Dane, na których pracuje aplikacja. Może to być plik TXT, AVI, HTML, JS, dane przesyłane po socketach, endpointach, argumenty funkcji etc. Generalnie dane aplikacji, na które ma wpływ użytkownik pracujący z daną aplikacją.
- Shellcode – Wstrzyknięty kod atakującego, który ma zostać wykonany podczas wykorzystania danej podatności.

Debugging

Debugging w pracy reversera/researchera, niezależnie od wykonywanego zajęcia (security research, malware/anti-malware, cracking/software protections, vulnerability research) jest podstawowym narzędziem/umiejętnością wykorzystywaną w czasie pracy. Pozwala odtworzyć logikę/zastosowane algorytmy programu bez potrzeby posiadania plików źródłowych aplikacji. W przypadku odnalezienia nowej podatności, pozwala na dokładną analizę problemu oraz jego potencjalne wykorzystanie. Implementując konkretny exploit musimy znać dokładny stan aplikacji w czasie wywołania podatności. Przez „stan" rozumiem stan całej przestrzeni adresowej procesu: stack(s), heap(s), thread(s), moduły wykonywalne załadowane w procesie, a w szczególności ich adresy oraz właściwości, ich tablice IAT (import)/EAT(export), systemowe struktury danych pozwalające systemowi operacyjnemu na zarządzanie procesem i wykonywaniem wątków etc.

Narzędzia

Win32 Debug API oraz CTYPES System operacyjny Windows posiada bogate API umożliwiające pisanie własnych debuggerów dla R3 (Ring 3 – User Mode) 1. Istnieje wiele powodów, dla których napisanie własnego debuggera jest często najrozsądniejszym rozwiązaniem. Debugger to nic innego jak swoisty nadzorca procesu, który może reagować, w zaprogramowany przez nas sposób, na zachodzące w procesie zdarzenia:

```
* CREATE_PROCESS_DEBUG_EVENT
* CREATE_THREAD_DEBUG_EVENT
* EXCEPTION_DEBUG_EVENT
* EXIT_PROCESS_DEBUG_EVENT
* EXIT_THREAD_DEBUG_EVENT
* LOAD_DLL_DEBUG_EVENT
* OUTPUT_DEBUG_STRING_EVENT
* UNLOAD_DLL_DEBUG_EVENT
* RIP_EVENT
```

Jednym z najciekawszych zdarzeń z naszej perspektywy jest EXCEPTION_DEBUG_EVENT. Zostaje on wywołany przez system w przypadku np.: dzielenia przez zero, dostępu do niezalokowanej pamięci, bądź zatrzymaniu się na ustawionym przez nas breakpointcie 2.

Exception codes 3:

```
* EXCEPTION_ACCESS_VIOLATION
* EXCEPTION_ARRAY_BOUNDS_EXCEEDED
* EXCEPTION_BREAKPOINT
* EXCEPTION_DATATYPE_MISALIGNMENT
* EXCEPTION_FLT_DENORMAL_OPERAND
* EXCEPTION_FLT_DIVIDE_BY_ZERO
* EXCEPTION_FLT_INEXACT_RESULT
* EXCEPTION_FLT_INVALID_OPERATION
* EXCEPTION_FLT_OVERFLOW
* EXCEPTION_FLT_STACK_CHECK
* EXCEPTION_FLT_UNDERFLOW
* EXCEPTION_ILLEGAL_INSTRUCTION
* EXCEPTION_IN_PAGE_ERROR
* EXCEPTION_INT_DIVIDE_BY_ZERO
* EXCEPTION_INT_OVERFLOW
* EXCEPTION_INVALID_DISPOSITION
* EXCEPTION_NONCONTINUABLE_EXCEPTION
* EXCEPTION_PRIV_INSTRUCTION
* EXCEPTION_SINGLE_STEP
* EXCEPTION_STACK_OVERFLOW
```

Dzięki własnemu debuggerowi, możemy w dowolny sposób nadzorować oraz modyfikować wykonywane w procesie wątki.

Z naszej perspektywy najbardziej interesujące są: * Odczytanie/modyfikacja danych w pamięci. * Nadzorowanie wyjątków i ich automatyczna analiza w przypadku sesji fuzzowania. * Modyfikacje wykonywanego kodu. * Monitorowanie wykonywania kodu. * Wstrzykiwanie kodu.

Dzięki wykorzystaniu CTYPES, w skryptach możemy wykorzystywać wywołania natywnych Win32 API w skryptach Python.

PyDBG PyDBG 4, stworzony przez Pedram Amini, jest niczym innym jak przyjemnym dla programisty wrapperem na Debug API napisanym w Pythonie. Umożliwia stworzenie małego, dedykowanego debuggera dosłownie w kilku linijkach kodu.

Funkcjonalność PyDBG: * Zarządzanie software breakpoints * Zarządzanie hardware breakpoints * Zarządzanie memory breakpoints * Zarządzanie uchwytami procesu * Zarządzanie i manipulacja rejestrami procesora * Zarządzanie modułami wykonywalnymi (ładowanie, odładowywanie) * Mapowanie adresów VA na załadowane moduły wykonywalne * Przetwarzanie debug events * Podłączanie się do procesu i odłączanie (ang. process attach/detach) * Disassemble * Dump context * Enumeracja modułów wykonywalnych w procesie * Enumeracja procesów i wątków * Tworzenie procesów * Tworzenie wątków w procesie * Kończenie procesów i wątków * Otwieranie procesów i wątków * Event handlery dla obsługi debug events * Obsługa łańcuchów znakowych znajdujących się w pamięci procesu * Zrzucanie snapshotu procesu * Odczyt/zapis pamięci procesu * Zatrzymywanie/wznawianie wątków w procesie * Zarządzanie stosami * Zarządzaniem stronami pamięci (zmiany flag: RWX)

PyDBG: Hooking PyDBG umożliwia wygodne ustawianie breakpointów dla wykonywanych w procesie wątków. Dzięki temu mamy łatwą możliwość odczytu argumentów funkcji, zmiennych globalnych, zmiennych lokalnych, buforów danych alokowanych na stosie bądź stercie (ang. heap) oraz oczywiście ich modyfikacji. Hooking umożliwia wygodne nadzorowanie pracy procesu oraz analizę działania wykonywanego kodu. Dzięki możliwości zaprogramowania obsługi każdego interesującego nas zdarzenia, możemy wprowadzić dowolną logikę, która pozwala nam analizować skomplikowane przetwarzanie danych przez dany wątek.

PyDBG: Dynamiczne modyfikowanie kodu Kolejnym krokiem, po odczytywaniu i modyfikowaniu danych używanych przez dany proces, jest nadzorowanie i modyfikacja kodu wykonywanego w danym wątku. Posiadamy możliwość reagowania na ustawione breakpointy, możemy dokonywać zmiany wartości rejestrów, zmiany wartości pamięci, zmieniać miejsce wykonania kodu, dokonywać zmiany decyzji o skokach warunkowych. Dodatkowo, jeśli sytuacja tego wymaga,

możemy wstrzyknąć nasz kod w postaci shellcodu, bądź dodatkowej biblioteki, w której znajduje się nasz kod.

Immunity Debugger Immunity Debugger 5 bazuje na debuggerze OllyDbg 6 stworzonym przez Oleh Yuschuk. Immunity Debugger rozszerza OllyDbg o interpreter Pythona wraz z pokaźnym API, umożliwiający tworzenie nowych narzędzi na powyższej platformie.

IDA & IDAPython Podobnie jak w przypadku Immunity Debuggera, stworzenie pluginu IDAPython 7 otworzyło środowisko IDA 8 na łatwe tworzenie nowych narzędzi w Pythonie, wspomagających analizę modułów wykonywalnych.

Budowa fuzzerów w Pythonie

Proces fuzzowania aplikacji można podzielić na kilka kroków: 1. Stworzenie danych wejściowych dla aplikacji na podstawie wzorca. 2. Uruchomienie aplikacji w nadzorowanym środowisku (np. z podłączonym debuggerem). 3. Załadowanie stworzonych w punkcie 1. danych do aplikacji. 4. Analiza zachowania aplikacji. a. Gdy aplikacja zachowuje się stabilnie (powrót do punktu 2.). b. Gdy aplikacja zachowuje się niestabilnie – zalogowanie błędu. 5. Końcowa analiza znalezionych błędów.

Każdy z tych punktów wymaga niezależnych narzędzi. Każde z narzędzi często wymaga też modyfikacji, w zależności od aplikacji, która ma być celem fuzzowania. Tworzenie danych wejściowych może być tak proste, jak tylko zmiany pojedynczych bitów/bajtów (dumb fuzzing) lub tak złożonym jak analiza danego formatu i zmiana tylko interesujących nas elementów (smart fuzzing). Niezależnie od wybranej strategii, generowanie danych za pomocą Pythona jest zadaniem stosunkowo łatwym. Stworzenie nadzorowanego środowiska wymaga dokładnej znajomości zagadnień z wielu dziedzin (protokołów sieciowych, architektury systemów operacyjnych, błędów bezpieczeństwa, reverse engineeringu, etc.). Najczęściej stosowanymi mechanizmami do nadzorowania aplikacji, jako procesu systemowego, są debuggery oraz monitory sieciowe (np. Wireshark). Tworzenie własnych debuggerów jest czasochłonne, ale pozwala na pełną kontrolę uruchomionej aplikacji. Dodatkowym atutem stosowania własnych debuggerów jest możliwość automatycznej analizy odnalezionego błędu, co skraca znacząco czas pracy researchera. Analiza wyników jest zadaniem niepowtarzalnym. Każda aplikacja/biblioteka może generować wyniki w innym formacie (core dump, plik tekstowy JVM, wyniki z własnego debuggera). Python jest doskonałym narzędziem do pisania własnych analizatorów plików wynikowych i wydobywania z nich istotnych szczegółów.

Sulley Sulley [9] jest frameworkiem przeznaczonym do fuzzowania aplikacji. Jest on w całości napisany w Pythonie. Jego niewątpliwymi zaletami są prostota

oraz kompleksowość. Pozwala on na tworzenie w intuicyjny sposób reprezentacji danych (request), które później są wykorzystywane w sesjach fuzzowania. Sulley zawiera wbudowane agenty, które pozwalają na monitorowanie środowiska: * network_monitor.py - logowanie komunikacji sieciowej do plików PCAP. * process_monitor.py - pozwala na monitorowanie fuzzowanego procesu. * etc. Dodatkowym atutem jest wbudowany serwer HTTP, pozwalający śledzić postępy fuzzowania. Jest to proste, ale potężne narzędzie, z którym definitywnie warto się zapoznać. Często może stanowić pierwszy krok w rozpoczęciu procesu testowania danej aplikacji. Sulley jest jednym z ważniejszych frameworkow wykorzystywanych przez security researcherów w ich pracy.

FuzzMyApp Fuzzing Framework „FFF" W FuzzMyApp rozwijamy własny framework, używany przez nas w czasie fuzzowania aplikacji. Powodem podjęcia takiej decyzji było ciągłe napotykanie powtarzalnych problemów, które wybitnie nadawały się do automatyzacji. Framework jest stworzony całkowicie w Pythonie 2.x. Jego głównymi zadaniami są: * Tworzenie zbioru danych wejściowych, dla zadanych strategii mutacji. * Uruchamianie aplikacji w nadzorowanym środowisku. * Agregacja i sortowanie wyników zebranych w czasie sesji fuzzowania. * Wstępna analiza odnalezionych podatności. * Generacja statystyk dla sesji fuzzowania. * Wielozadaniowość.

Wielozadaniowość w naszym kontekście oznacza dany scenariusz testowy. Czyli przykładowo fuzzowanie danego formatu pliku dla konkretnej aplikacji. System stworzony jest w taki sposób, by wykonać jak najwięcej powtarzalnych zadań, pozostawiając researcherowi/reverserowi analizę odkrytych podatności. Większość tych zadań nie jest krytyczna czasowo, wiec zdecydowaliśmy się właśnie na Pythona, w celu szybkiej i przyjemnej implementacji naszego frameworku. Decyzja o wykorzystaniu Pythona została również podjęta dzięki tak dużej liczbie zewnętrznych narzędzi/bibliotek, które znacząco ułatwiają pracę. Framework został napisany w sposób modułowy, dzięki czemu jego dostosowanie do nowej aplikacji wymaga minimalnego nakładu pracy.

Fuzzowanie aplikacji napisanych w Javie Jednym z napotkanych problemów podczas pracy w FuzzMyApp było stworzenie narzędzi do fuzzowania aplikacji napisanych w Javie. Proces ten podzieliliśmy na dwie części: 1. Znalezienie potencjalnych niebezpiecznych metod. 2. Stworzenie kodu fuzzującego dla metod znalezionych w punkcie 1.

Punkt 2 został rozwiązany poprzez stworzenie dedykowanego generatora kodu Javy, który importował interesujące nas klasy i fuzzował wskazane metody. Generacja kodu Javy oraz jego kompilacja została zaimplementowana jako skrypt Pythona, bazujący na pliku wzorcowym. Sam proces fuzzowania odbywał się poprzez uruchomienie wszystkich wygenerowanych plików klas. Dodatkowym ułatwieniem było samo zachowanie JVM, która zrzuca bardzo opisowe tekstowe pliki dump w wypadku nieobsłużonego błędu w samej wirtualnej maszynie Javy.

Aby znaleźć metody, które moglibyśmy poddać fuzzowaniu, postanowiliśmy statycznie analizować bytecode Java. W tym celu stworzyliśmy dedykowane narzędzie w Pythonie, w oparciu o dokumentację specyfikacji JVM. Do tego zadania doskonale nadaje się moduł struct. Naszym celem nie było odtworzenie pełnej specyfikacji, lecz tylko stworzenie narzędzia, pozwalającego wydobyć sygnatury metod dostępnych w danej klasie. Dzięki tym sygnaturom mogliśmy przefiltrować listę dostępnych metod i wybrać tylko te, które: a. przyjmowały parametry (metody bezparametrowe nie mogą być fuzzowane). b. były metodami publicznymi (brak potrzeby wykorzystania refleksji). c. były metodami zaimplementowanym natywnie (słowo kluczowe native) przy pomocy JNI.

Takie podejście okazało się dużym sukcesem. Udało nam się znaleźć błędy bezpieczeństwa m.in. w JRE7 oraz JOAL: * FMA-2013-010 10 – wiele błędów w JRE7. Oracle nie uznał ich jako błędy bezpieczeństwa, w przeciwieństwie do firmy specjalizującej się w Vulnerability Research - Beyond Security [11]. * FMA-2012-038 [12] (CVE-2013-4099) – prawie 70 błędów Remote Code Execution w projekcie JOAL (OpenAL.dll). Nasza praca doprowadziła do największego przeglądu bezpieczeństwa projektu [13].

Wykorzystanie Pythona pozwoliło na pełną automatyzację procesu, którego czas oscylował wokół kilkunastu godzin ciągłej pracy.

Emulacja CPU IA-32 - PyEmu

PyEmu [14] jest emulatorem dla procesora o architekturze IA-32 napisanym w Pythonie. Wprowadza abstrakcję na procesor oraz pamięć procesu. Podobnie jak Immunity Debugger oraz IDAPython dla produktu IDA, stanowi kolejny framework umożliwiający tworzenie nowych narzędzi do dynamicznej analizy kodu w oparciu o warstwę abstrakcji, jaką jest emulator procesora.

Podsumowanie

Niniejszy artykuł jest tylko wstępem do problemu wyszukiwania błędów bezpieczeństwa w aplikacjach, frameworkach, bibliotekach i samych systemach operacyjnych. Jest to zagadnienie pełne problemów i wyzwań, w których znacząco pomaga zastosowanie Pythona jako języka programowania. Dodatkowym atutem jest liczna baza narzędzi zaimplementowanych i dostępnych publicznie w tym języku. Narzędzia napisane w Pythonie w łatwy sposób można szybko modyfikować i przystosowywać do nowych zastosowań. Python jest fantastycznie prostym, czytelnym, jednakże potężnym językiem programowania. Pisząc obiektowo/proceduralnie w innych językach może nam brakować wielu elementów w Pythonie, jednak przeważnie znajdzie się sposób na ich analogiczną reprezentację. Python umożliwia budowanie i utrzymywanie nie tylko prostych skryptów ale i dużych obiektowych rozwiązań. Jednak przy dużych rozwiązaniach warto trzymać się sprawdzonych procesów produkcji i utrzymania oprogramowania i

... pisać dużo testów jednostkowych! Czasami pełen dynamizm oraz prostota w Pythonie może być negatywnym czynnikiem, jeśli mamy duże rozwiązanie z n warstwami abstrakcji. Brak prostych mechanizmów definiowania zakresu widoczności bądź wymuszania implementacji, innych niż rzucanie wyjątkami w czasie runtime. Jednak powyższe aspekty (zalety/wady) Pythona pozostawmy na kuluarowe dyskusje, do których serdecznie zapraszamy.

Referencje

- 1 Linki MSDN Debugging Reference http://msdn.microsoft.com/en-us/library/windows/desktop/ms679304(v=vs.85).aspx
- 2 Structured Exception Handling http://msdn.microsoft.com/en-us/library/windows/desktop/ms680660(v=vs.85).aspx
- 3 SEH Exception Codes http://msdn.microsoft.com/en-us/library/windows/desktop/aa363082(v=vs.85).as
- 4 PyDBG https://github.com/OpenRCE/pydbg
- 5 Immunity Debugger http://www.immunityinc.com/products/debugger
- 6 OllyDbg http://www.ollydbg.de
- 7 IDAPython http://code.google.com/p/idapython
- 8 IDA https://www.hex-rays.com
- [9] Sulley https://github.com/OpenRCE/sulley
- 10 http://www.fuzzmyapp.com/pl/advisories.html
- [11] http://www.securiteam.com/securitynews/5DP3K0AAVA.html
- [12] http://www.fuzzmyapp.com/advisories/FMA-2012-038/FMA-2012-038-PL.xml
- [13] https://www.jogamp.org/blog/index1.html
- [14] PyEmu http://code.google.com/p/pyemu

Common programming mistakes in Python

Dariusz Śmigiel

Introduction

Python is known as one of the simplest programming languages. You need to know syntax, some basic things like intendation and voilà! You're a developer! But... not so fast. Often it turns out that that's not everything that you need to know about Python to be an efficient developer. You need to know some subtle things. Without this, you can have big problems, because behaviour that you have, is different than expected.

Full disclosure Idea for this talk was taken from work done by Martin Chikilian, originally published at www.toptal.com/python/top-10-mistakes-that-python-programmers-make

Mistake #1: Expressions as defaults for function arguments

Python allows to specify default values for function arguments. When function is called without the argument, argument will have assigned value provided as default. It's a big advantage, because user doesn't have to remember, or even know, about providing values to function. It works pretty good, as long, as you won't give any mutable values there.

```
>>> def foo(bar=[]):
...     bar.append("baz")
...     return bar
```

Current behaviour It's simple function, where we want to append simple string to list. Expected behaviour of this function, would be:

- create list called `bar`
- append to it a string `baz`
- return list

Unfortunately, it doesn't work like expected. When function is called, we're receiving the same list, with growing number of strings.

```
>>> foo()
['baz']
>>> foo()
['baz', 'baz']
>>> foo()
['baz', 'baz', 'baz']
```

What's happening? List `bar` is initialized only once; when function definition is evaluated. That's why, when we're calling `foo`, every time we're appending new string to the same list. This behaviour is implemented on purpose. Early binding means that the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address.

Solution

```
>>> def foo(bar=None):
...     if bar is None:
...         bar = []
...     bar.append("baz")
...     return bar

>>> foo()
['baz']
>>> foo()
['baz']
```

Mistake #2: Binding variables in closures

This time, we'll look at late bindings. Assume, we have function to build 5 next multipliers of given value.

```
>>> def create_multipliers():
...     return [lambda x : i * x for i in range(5)]
>>> for multiplier in create_multipliers():
...     print(multiplier(2))
...
```

Current behaviour We're expecting to retrieve:

```
0
2
4
6
8
```

But, instead of this we've got:

```
8
8
8
8
8
```

What's happening? Closures are *late binding*. The values of variables used in closures are looked up at the time the inner function is called. Whenever any of the returned functions are called, the value of `i` is looked up in the surrounding scope at call time. By then, the loop has completed and `i` is left with its final value of 4. And, contrary to popular belief, it has nothing to do with *lambdas*.

```
>>> def create_multipliers():
...     multipliers = []
...
...     for i in range(5):
...         def multiplier(x):
...             return i * x
...         multipliers.append(multiplier)
...     return multipliers
...
>>> for multiplier in create_multipliers():
...     print(multiplier(2))
...
...
8
8
8
8
8
>>>
```

Solution As mentioned in #1, Python evaluates early the default arguments of a function, so we can create a closure that binds immediately to its arguments by using default arg:

```
>>> def create_multipliers():
...     return [lambda x, i=i : i * x for i in range(5)]
...
>>> for multiplier in create_multipliers():
...     print (multiplier(2))
...
...
0
2
4
6
8
>>>
```

But, even better idea is change it to explicit:

```
>>> def get_func(i):
...     return lambda x: i * x
...
>>> def create_multipliers():
...     return [get_func(i) for i in range(5)]
...
>>> for multiplier in create_multipliers():
...     print (multiplier(2))
...
...
0
2
4
6
8
>>>
```

Mistake #3: Local names are detected statically

Again, our main concern is about closures. We want to print global `x`, and later change it to new value.

```
>>> x = 99
>>> def func():
...     print(x)
...     x = 88
...
```

Current behaviour And, again. Something goes wrong.

```
>>> func()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in func
UnboundLocalError: local variable 'x' referenced before assignment
```

What's happening While parsing this code, Python sees the assignment to `x` and decides that `x` will be a local variable in the function. But later, when the function is actually run, the assignment hasn't yet happened when the print executes, so Python raises an undefined name error.

Solution In this case, we have two solutions: nasty (`global`) and better (reference it through enclosing module name)

```
>>> x = 99
>>> def func():
...     global x
...     print(x)
...     x = 88
...
>>> func()
99
```

Mistake #4: Class variables

We have three classes:

```
>>> class A(object):
...     x = 1
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
```

Current behaviour After initialization, we're able to get `x` from all classes

```
>>> print(A.x, B.x, C.x)
1 1 1
```

We can also assign variables:

```
>>> B.x = 2
>>> print(A.x, B.x, C.x)
1 2 1
```

And another time:

```
>>> A.x = 3
>>> print(A.x, B.x, C.x)
3 2 3
```

Oops.

What's happening? MRO is happening. Because `C` class has no attribute `x`, it goes to class `A` and returns value for it.

Mistake #5: Exception handling

Python has nice feature to catch any raised exception. It allows you to prevent the software from crashing, reacting for expected or non-expected situations. In this case, we're using `try/except` clause:

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])
... except ValueError, IndexError:
...     pass
```

Current behaviour Our code seems to be very well protected. In one place we're catching value conversion problem and also index bigger than list. Unfortunately, it doesn't work like we would like to:

```
...
Traceback (most recent call last):
  File "<input>", line 3, in <module>
IndexError: list index out of range
```

What's happening? In Python 2 old syntax is still supported for backwards compatibility. In modern Python, we would write: `except ValueError as e` what is equal to `except ValueError, e` In our case, `except ValueError, IndexError` is equivalent to `except ValueError as IndexError` which is not what we want.

Solution Python 3 has no problems with above code. It throws an error, and shows exact place, where something doesn't work:

```
File "<stdin>", line 4
  except ValueError, IndexError:
```

But if you're still using Python 2 and don't see any warning signs, you'll need to follow below solution:

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])
... except (ValueError, IndexError) as e:
...     pass
```

Works OK, for both, Python 2 and Python 3.

Mistake #6: Scope rules

Sometimes, we want to know, how many times some code was run. It would be good, to count it. So, we'll write a simple snippet:

```
>>> bar = 0
>>> def foo():
...     bar += 1
...     print(bar)
...
```

Current behaviour And run it:

```
>>> foo()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in foo
UnboundLocalError: local variable 'bar' referenced before assignment
```

BAM! The same happens, when we change line `i += 1` to `i = i + 1` It doesn't matter which one is used, both throw an error.

What's happening? Python scope resolution is based on LEGB:

- Local - names assigned in any way within a function (*def* or *lambda*) and not declared global in that function;
- Enclosing function locals - name in the local scope of any and all enclosing (*def* or *lambda*), from inner to outer;
- Global (module) - names assigned at the top-level of a module file, or declared global in a *def* within the file;
- Built-in (Python) - names preassigned in a built-in names module: *open*, *range*, *SyntaxError*, ...

So, when you make an assignment to variable, Python considers it as in local scope. It shadows everything, that is outside this scope. In this case, we don't *see* variable `i` declared before function definition.

Solution Variable needs to be modified in the scope that it was declared in. We can achieve this by passing it as an argument to a function that will return it's new value.

```
>>> bar = 0
>>> def foo(bar=bar):
...     bar += 1
...     return bar
...
>>> foo()
1
>>> foo()
1
```

Mistake #7: Modifying list while iterating over it

Often we need to do something with lists. For instance, remove all odd values from list.

```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> for i in range(len(numbers)):
...     if odd(numbers[i]):
...         del numbers[i]
...
Traceback (most recent call last):
      File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

What's happening We're iterating over this list, and when value is odd, we're removing it from list. In the same time list is shrinking, so after few iterations, list is shorter than expected.

Solution Suggested solution, probably the simplest one.

```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> for n in numbers:
...     if odd(n):
...         numbers.remove(n)
...
>>> numbers
[0, 2, 4, 6, 8]
```

But, we cannot fall into false impression, that this works.

```
>>> odd = lambda x: bool(x % 2)
>>> numbers
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9]
>>> for n in numbers:
...     if odd(n):
...         numbers.remove(n)
...
>>> numbers
[0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9]
```

As you may see, it iterates, removes, but also leaves odd numbers. Does it mean, there is no proper solution for it? No! To solve it, we will use list comprehensions

```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> numbers = [n for n in numbers if not odd(n)]
>>> numbers
[0, 2, 4, 6, 8]
```

Mistake #8: Name clashing with Python Standard Library modules

Sometimes, we forget about simple things. Not every possible module name should be used. Assume, you're creating an application for sending emails.

```
app
|-sender.py
|-receiver.py
|-email.py
```

```
sender.py

from email.mime.multipart import MIMEMultipart

msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = 'from@email.com'
msg['To'] = 'to@email.com'
```

Current behaviour When we're trying to run this application, we've receiving `ImportError`

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ImportError: No module named mime.multipart
```

What's happening We've mixed Standard Library module called `email` with local module `email.py`. In this case, application sees local module, and imports it, instead of expected one.

Solution In this case, we have to be very careful. Python uses pre-defined order of importing modules. When we're trying to import `spam` it looks in:

- built-in module (string, re, datetime, etc.)
- searches for a file named `spam.py` in directories given by the variable `sys.path` in
- the directory containing the input script (or the current directory).
- PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
- the installation-dependent default.

Bonus Mistake: Differences between Python 2 and Python 3:

Simple thing can make a big difference. Python 3 handles exception in local scope.

```
import sys

def bar(i):
    if i == 1:
        raise KeyError(1)
    if i == 2:
        raise ValueError(2)

def bad():
    e = None
    try:
        bar(int(sys.argv[1]))
    except KeyError as e:
        print('key error')
    except ValueError as e:
        print('value error')
    print(e)

bad()
```

Current behaviour `Python 2`

```
$ python foo.py 1
key error
1
$ python foo.py 2
value error
2
```

```
Python 3
```

```
$ python3 foo.py 1
key error
Traceback (most recent call last):
  File "foo.py", line 19, in <module>
bad()
  File "foo.py", line 17, in bad
print(e)
UnboundLocalError: local variable 'e' referenced before assignment
```

What's happening When an exception has been assigned to a variable name using `as target`, it is cleared at the end of the except clause:

```
except E as N:
    try:
        foo
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the except clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Designing an API is an incredibly important moment for a project's future. Every change made in the future will be accompanied by having to support multiple versions of an API, modifying docs, server-side code and the client. Because of this laundry list of changes to keep track of, we try to solve as many design problems as possible up front.
There are several different areas, that make a huge impact. The one we target in this post is latency.

> Latency is a time interval between the stimulation and response (. . .).
> Wikipedia, Latency (engineering)

What is latency when it comes to mobile devices? Mobile networks are still far from what we would call high performance. There is huge overhead required when you make even a single request from your mobile app. A quick list of the steps needed:

- RRC negotiation
- DNS lookup
- TCP handshake
- TLS handshake
- HTTP request

Beyond just this one complicated path, we need to keep in mind some potential problems:

- weak signal
- lost signal
- client is moving and changing base stations
- routing gets mixed up - source
- different pings originating from the same location(home/office) - source

You end up with a latency of 100ms-300ms for 3G networks and up to 1000ms for 2G. Since 2G is becoming obsolete and 4G is just around the corner for our users (check this thread), let's take a look at a forecast from Cisco Systems, Inc. (source).

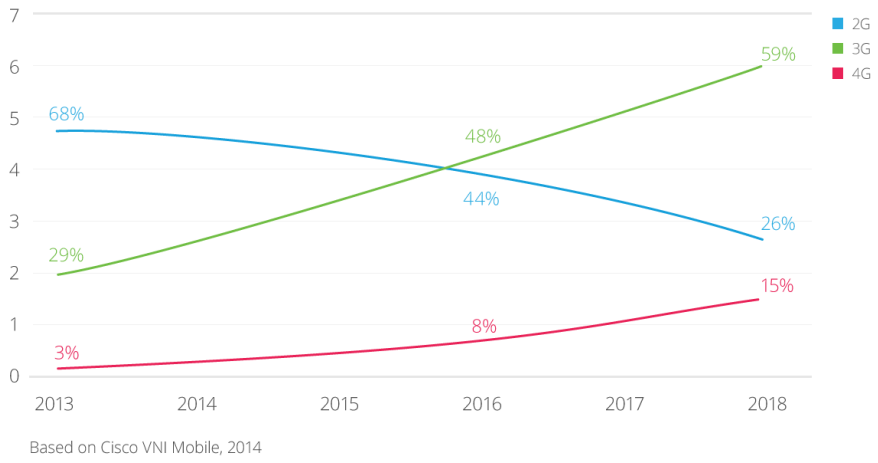

Figure 1: CISCO mobile connection speed forecast

What this shows is that 4G shouldn't be treated as the representative network connection of our users. Of course we should be able to target our application well and if it's an app for geeks, let's say a StackOverflow client, the subsequent network connection speed will be drastically higher than when compared to a general purpose app like a news reader.

We can deal with latency by trying to optimize DNS lookup counts, shorten the request route, and many many more. We could even go so far as to spend 25 million dollars to improve it by another 1ms. However, this article is addressing a rather specific part connected to API design. So, how can we influence the latency by API design?

Merging responses

First, let's make a use of an example. Let's say we have a view that says user name, age and country. Name and age are a part of `/users/{id}` resource:

```
{
  "name": "John Doe",
  "age": 25,
  "links": [
    {
      "rel": "self",
      "href": "/users/123"
    },
    {
      "rel": "account",
      "href": "/accounts/987"
    },
    {
      "rel": "address",
      "href": "/users/123/address"
    }
  ]
}
```

To display a country, we need to make another call, for `/users/{id}/address`. This means following the entire route to reach the server again:

```
{
  "street": "Sesame",
  "no": 25,
  "zipCode": "12-321",
  "state": "NY",
  "country": "US"
}
```

But since there's no real reason to make both of those calls, let's try to save one connection and prepare a special endpoint, dedicated for that screen:

```
{
  "name": "John Doe",
  "age": 25,
  "country": "US"
}
```

Now instead of making two separate calls we may just call a single endpoint and get all of the required information. But remember that there is a tradeoff. To get a more robust API we're stepping away from a REST design pattern and binding together two related entities - user and address.

Expansion

We don't always know the way in which our API will be used. In the last example we implicitly assumed that we knew what information was being displayed. But what about cases where we design a service for future consumers which may present data in any way they decide? Yet we still want to give the option to optimize answers.

In that case an expansion pattern becomes very handy. Going to our previous example, what would this look like? We need to prepare a request that specifies which fields to display (and embed from related entities). `GET /users/123` now becomes `GET /users/123?fields=[name,age,address[country]]`.

```
{
  "name": "John Doe",
  "age": 25,
  "address": {
    "country": "US"
  }
}
```

This is a pretty neat mechanism for merging responses, although it has it's own limitations as well. If API consumers are using relations, we won't be able to change the resource hierarchy without breaking clients.

Keep-alive

The closing trick is to make sure your service supports the HTTP keep-alive option. By default, all mobile HTTP clients will try to sustain the HTTP connection between different calls so it makes perfect sense to utilize this. If the server will agree to keep the connection, the next requests will omit several handshakes and directly be sent to server within an opn connection.

Closing word

In Polidea we are developing mobile apps and supporting them with backend services. The last couple of years has taught us that you may not achieve a proper user experience without slick HTTP communication. This article was inspired by our experiences from real-world projects. As always, we welcome any feedback and would be happy to discuss this topic as well as our work in further details.

Python w Chmurach

Dariusz Aniszewski

Wyobraźmy sobię osobę związaną z branżą IT, która nie spotkała się jeszcze z terminem *chmura obliczeniowa*. Nie da się. Słowa *as a Service* robią ogromną karierę i cieszą się rosnącą popularnością. Z chmur korzysta coraz więcej użytkowników, od małych developerów po wielkie korporacje. Jeśli ktoś myśli, że chmury obliczeniowe nie są dla niego, najprawdopodobniej jest w błędzie. Jeśli ktoś uważa, że nie używa ich na co dzień, również najprawdopodobniej nie ma racji. Serwisy takie jak Facebook albo Gmail to typowe usługi chmurowe. Dropbox i Dysk Google to niemal książkowe przykłady chmury oferującej usługę przechowywania danych. Przykłady można mnożyć. Skupmy się jednak na tym, jak wykorzystać *cloud computing* do rozwijania własnych Pythonowych aplikacji. Na początek jednak krótka nota historyczna.

Krótka historia

Mogłoby się wydawać, że chmury obliczeniowe to wynalazek ostatnich kilku lat. A jednak... Ich idea sięga końca lat 60-tych ubiegłego wieku i zakłada istnienie rozproszonych, skalowalnych systemów, dostępnych z dowolnego miejsca na świecie. Przez lata jednak niewiele się działo w tym kierunku, przynajmniej w sferze rozwiązań dostępnych publicznie. W 1999 roku firma Salesforce zaczęła dostarczać swoje produkty przez przeglądarkę - był to kamień milowy i prekursor do modelu *Software as a Service*. Następny kamień położył Amazon, startując w 2002 roku z Amazon Web Services, a 4 lata później udostępniając usługę *Elastic Cloud Computing* (EC2) będącą udzieleniem dostępu do maszyny wirtualnej pracującej w chmurze. Od tego dnia Internet nie był już taki sam. W 2009 roku do wyścigu dołączył Google, a chwilę później Microsoft. Następnie nastąpił boom na różne usługi dostępne w modelu chmury obliczeniowej i trwa on po dzień dzisiejszy.

Chmura obliczeniowa

Czym więc jest w obecnym czasie chmura? Najprościej - usługą. Usługą przechowywania plików, albo usługą serwisu bazodanowego. Może to też być usługa udostępnienia maszyny wirtualnej, albo przeprowadzania obliczeń. Może to być jakakolwiek usługa, możliwa do dostarczenia przez Internet, za jaką użytkownicy będą skłonni zapłacić. Obecnie chmury możemy podzielić na trzy typy:

- Infrastructure as a Service (IaaS) - w tym modelu usługą jest najczęściej maszyna wirtualna
- Platform as a Service (PaaS) - dostęp do platformy, na której możemy uruchamiać aplikacje
- Software as a Service (SaaS) - oprogramowanie dostępne przez przeglądarki

Idąc po kolei, w modelu *IaaS* klient dostaje dostęp do pewnej części infrastruktury teleinformatycznej. Najczęściej jest to po prostu maszyna wirtualna, ale mogą to być też np. serwery SMTP, czy baza danych. Konfiguracja i instalacja oprogramowania spoczywa na kliencie. Model *PaaS* jest rozwinięciem infrastruktury, w której operator dostarcza już pewne mechanizmy odciążające użytkowników z instalacji oprogramowania, czy konfiguracji serwerów. Klient musi jedynie wgrać swoją aplikację oraz wskazać plaftormie sposób jej uruchomienia. *SaaS* jest natomiast niczym innym jak oprogramowaniem dostępnym przez przeglądarkę. Oprogramowanie tego typu jest bardzo zróżnicowane - mogą to być pakiety biurowe, programy graficzne, księgowe i wiele innych.

Koszty usług chmurowych są bardzo zróżnicowane i każdy operator ma swój cennik za poszczególne produkty. Panuje jednak jedna zasada - klient jest rozliczany z używanych zasobów. Dla przykładu jeśli trzeba przeprowadzić skomplikowane obliczenia, można utworzyć kilka maszyn wirtualnych, połączyć je w macierz i po zakończeniu obliczeń je wyłączyć. Koszty będą generowane tylko podczas pracy tych maszyn. Jest to ogromna zaleta chmur, bez których konieczny byłby zakup fizycznych komputerów, które byłyby wykorzystywane sporadycznie, a koszt ich zakupu i konserwacji znacząco przewyższałby koszty takiego samego zestawu w chmurze.

Przygotowania

Skoro wiemy już co nieco o chmurach, czas je wykorzystać w praktyce. W chmurze możemy serwować wszelkie aplikacje Pythonowe posiadające wywołania po protokole HTTP. Możemy zatem użyć dowolnego frameworka do aplikacji internetowych, np. Django, czy też Flask. Możemy również sami zaimplementować nasz serwer używając np. modułu *SimpleHTTPServer* z biblioteki standardowej. Dla dalszych rozważań przyjmijmy, że nasza aplikacja będzie korzystała z Django. Ta aplikacja będzie korzystała z relacyjnej bazy danych PostgreSQL

(choć może być też np. MySQL) i będzie umożliwiała użytkownikom wgrywanie plików do 30MB. Komunikację będziemy prowadzili po protokole HTTPS. Mamy wykupioną własną domenę oraz certyfikat SSL. Spodziewamy się umiarkowanego ruchu, który z czasem będzie wzrastał. Dobrze i co dalej?

Musimy teraz wybrać typ chmury, z jakiej chcemy korzystać, oraz jej dostawcę. Dla naszej aplikacji rozważmy wady i zalety modeli IaaS oraz PaaS. Jako reprezentantów obu modeli przyjmijmy: Amazon EC2 dla infrastruktury oraz Heroku dla platformy. Sprawdźmy najpierw, jak dokładnie działają obydwa te serwisy.

Amazon EC2

Usługa Elastic Cloud Computing jest niczym innym, jak udostępnianiem wirtualnych maszyn. Możemy zatem utworzyć maszynę posiadającą określoną moc obliczeniową, czy dysk twardy. Dodatkowo wybieramy system operacyjny, jaki ma na niej zostać zainstalowany oraz generujemy parę kluczy RSA, potrzebnych do połączenia się z maszyną przez SSH. Jeśli chcemy wykorzystać swój zestaw kluczy, możemy podać swój klucz publiczny zamiast generować nową parę. W obecnej chwili możemy wybierać z systemów UNIXowych, np. Ubuntu, Debian, CentOS oraz Microsoft Server. Każda maszyna przy uruchomieniu ma przyznawany publiczny adres IP, można też zarezerwować stały adres IP i przypiąć go do danej maszyny, wtedy pomimo restartów publiczny adres IP nie ulegnie zmianie. Po utworzeniu maszyny i przyznaniu jej stałego adresu musimy przystąpić do konfiguracji, czyli m.in.:

- otworzyć porty 22 (SSH), 80 (HTTP), 443 (HTTPS), ewentualnie inne wedle potrzeby
- zainstalować wymagane komponenty systemowe
- zainstalować i skonfigurować niezbędne serwisy
- zainstalować wymagane paczki Pythonowe
- wgrać, skonfigurować i uruchomić naszą aplikację i wszystkie jej komponenty

Jak widać, na uruchomienie aplikacji na EC2 należy poświęcić nieco czasu i pracy, jednak w zamian mamy pełną kontrolę nad tym, co się właściwie dzieje. Jedyne ograniczenia wynikają głównie z możliwości utworzonej maszyny.

Heroku

Heroku jest platformą uruchomieniową dla aplikacji napisanych w różnych językach - Ruby, Python, Java, NodeJS i innych. Heroku do swojego działania wykorzystuje wirtualizowane kontenery UNIXowe, marketingowo zwane *Dyno*. Dla uproszczenia można przyjąć, że jedno *Dyno* to odpowiednik jednej

maszyny, choć w praktyce na jednej wirtualnej maszynie może istnieć wiele wyizolowanych kontenerów. Aby uruchomić aplikację na Heroku, należy poinformować platformę, jak należy to zrobić. W przypadku Pythona należy wykonać dwa kroki:

1. dostarczyć plik **requirements.txt** z wylistowanymi paczkami Pythonowymi, jakie muszą zostać zainstalowane
2. dostarczyć plik **Procfile**, w którym podamy komendę uruchamiającą aplikację, np. `web: python manage.py runserver 0.0.0.0:$PORT`

Aby plafroma Heroku mogła sprawnie i bezobsługowo działać, narzucone są pewne ograniczenia:

- ulotny system plików - jeśli w trakcie działania naszej aplikacji coś zostanie zapisane na dysk, w momencie jej restartu będzie bezpowrotnie utracone. Jest to bardzo ważne ograniczenie, które należy mieć w świadomości już na etapie projektowania aplikacji.
- 30 sekundowy timeout - jeśli nasza aplikacja nie wyrobi się w ciągu 30 sekund z rozpoczęciem wysyłania odpowiedzi, Dyno Manager zwróci status 503. Należy mieć na uwadze, że pomimo wysłania statusu błędu, Dyno będzie ciągle przetwarzać zapytanie, w tym przeprowadzać rozpoczęte operacje na bazie danych, co może prowadzić do niespójności.
- 512MB pamięci operacyjnej - jeśli aplikacja potrzebuje dużej ilości pamięci operacyjnej, Heroku nie jest najlepszym wyborem - po przekroczeniu 512MB Heroku zacznie używać pliku wymiany (SWAP), co drastycznie wpłynie na wydajność.
- usypianie Dyno - jeśli nasza aplikacja wykorzystuje tylko jedno Dyno i w ciągu godziny nie było wykonane żadne zapytanie, Dyno zostaje uśpione. Uśpione Dyno nie pracuje, więc jeśli zostanie wysłane do niego zapytanie to Dyno Manager musi je najpierw wybudzić, co powoduje, że pierwsze zapytania mogą mieć dłuższy czas oczekiwania na odpowiedź.

Niewątpliwym plusem Heroku jest natomiast minimalna ilość pracy, jaką należy wykonać, aby uruchomić aplikację. Dodatkowo Heroku oferuje bardzo dużo dodatków, np.:

- bazy danych: relacyjne i NoSQL
- monitoring
- analiza wydajności
- agregowanie i przeszukiwanie logów
- i wiele innych

Te dodatki są de facto powiązanymi usługami chmurowymi, którymi możemy rozwijać naszą aplikację.

Aplikacje na Heroku mają swoje własne domeny w postaci `<nazwa-aplikacji>.herokuapp.com` działające zarówno po HTTP jak i HTTPS. Dostajemy również repozytorium Git, służące do wgrywania kolejnych wersji.

Wybór chmury

Skoro wiemy już czym charakteryzują się EC2 i Heroku, nadszedł czas na wybranie najbardziej pasującej usługi. W tym celu dokonajmy konfrontacji wymagań naszej aplikacji z możliwościami serwisów.

Wsparcie dla Django

Zarówno Heroku jak i EC2 mają wsparcie dla aplikacji korzystających z frameworka Django. Przy EC2 należy poświęcić więcej czasu na instalację i konfigurację systemu niż ma to miejsce na Heroku, gdzie musimy tylko dostarczyć plik requirements.txt (który prawdopodobnie i tak mamy) i Procfile. Na EC2 sami musimy zainstalować paczki Pythonowe wraz z wymaganymi pakietami systemowymi.

Baza danych PostgreSQL

W kwestii bazy danych sytuacja jest bardziej zawiła, ale oba serwisy umożliwiają nam jej obsługę. Wybierając ofertę Amazona mamy do wyboru dwie opcje. Pierwszą jest instalacja serwera bazodanowego na tej samej maszynie, na której działa nasza aplikacja. Drugą opcją jest skorzystanie z dedykowanej bazom danych usługi *Amazon Relational Database Service* - **RDS**. Wybór wariantu nie jest prosty - w przypadku instalacji na tej samej maszynie będziemy mieli problemy ze skalowalnością, ale korzystając z RDS generujemy dodatkowe koszta w przypadku używania tylko jednej instancji EC2 dla obsługi naszej aplikacji.

Na Heroku sprawa jest rozwiązana za nas. Baza danych dostępna jest jako dodatek, a my musimy jedynie wybrać plan cenowy. Na początek wystarczy plan darmowy, ale posiada on limit 10 000 wierszy. Po przekroczeniu tej liczby baza przestanie dodawać nowe rekordy i będziemy zmuszeni do zmiany planu na płatny.

Wgrywanie plików do 30 MB

To jest kluczowy punkt w specyfikacji wymagań. Umożliwienie wgrania plików przez użytkowników oznacza, że musimy je gdzieś przechowywać, a ich rozmiar będzie rósł wraz z liczbą użytkowników. Jeśli pamiętamy ograniczenia zarówno EC2 jak i Heroku, dochodzimy do wniosku, że żadne z nich nam nie wystarczy. Jeśli wybierzemy EC2 to prędzej czy później wyczerpie nam się miejsce na dysku. W przypadku Heroku system plików jest ulotny, więc wgrane pliki są

nietrwałe. Rozwiązaniem obydwu problemów będzie skorzystanie z kolejnej usługi - *Amazon Secure Storage Service* (Amazon **S3**). Ta usługa jest również polecana przez Heroku do przechowywania plików. S3 jest, jak nazwa mówi, usługą przechowywania plików, w której miejsce jest nieograniczone. Rozliczani jesteśmy z ilości danych, jakie przechowujemy oraz transferu. Do wgrywania plików na S3 możemy użyć modułu `boto`.

Obsługa HTTPS na własnej domenie

Obydwa serwisy umożliwiają bezpieczną komunikację. Na EC2 należy wgrać certyfikaty na serwer WWW i odpowiednio go skonfigurować. Niestety na Heroku, chcąc korzystać z HTTPS z własną domeną, poza wgraniem certyfikatów jesteśmy zmuszeni do dokupienia dodatku *SSL Endpoint* w cenie 20$/miesiąc.

Skalowalność

Skalowalność jest bardzo ważnym aspektem aplikacji, obydwa serwisy oferują wsparcie. Tak samo, jak w przypadku innych wymagań, zapewnienie skalowalności na EC2 jest bardziej złożone i wymaga dodania maszyny typu *Load Balancer* oraz utworzenia żądanej liczby maszyn z pracującą aplikacją. Na Heroku można zmienić liczbę używanych Dynos korzystając z przeglądarki i ustawiając żądaną liczbę w konfiguracji aplikacji.

Konkluzja

Obydwa serwisy jak najbardziej nadają się do serwowania aplikacji Pythonowych, a ich różnice wynikają w dużej mierze z wyboru modelu, w jakich pracują. W ogólnym rozliczeniu EC2 w modelu IaaS będzie wypadało korzystniej pod kątem miesięcznych opłat, lecz będzie wymagało większych nakładów na konfigurację i utrzymanie systemu. Heroku jest wygodniejsze w obsłudze, jednak w środowisku produkcyjnym zapewne będzie droższe od EC2.

Podsumowanie

Jak widać wybór serwisu chmurowego dla naszej aplikacji nie jest sprawą prostą i wymaga bardzo indywidualnego podejścia. Pod uwagę należy wziąć zarówno obecne wymagania funkcjonalne i pozafunkcjonalne oprogramowania, perspektywy rozwoju oraz budżet, jakim dysponujemy.

Źródła

- http://www.citeworld.com/article/2114518/cloud-computing/saas-top-50-list.html - artykuł wymieniający serwisy chmurowe używane na co dzień.
- http://www.computerweekly.com/feature/A-history-of-cloud-computing - historia chmur obliczeniowych
- https://aws.amazon.com/marketplace/search?page=1&category=2649367011 - dostępne systemy na EC2
- https://devcenter.heroku.com/categories/language-support - języki programowania wspierane na Heroku
- https://devcenter.heroku.com/articles/dynos - opis kontenerów Dyno
- https://addons.heroku.com/ - dodatki do Heroku
- Wykorzystanie technologii chmury obliczeniowej do zwiększenia zasięgu oprogramowania desktopowego - praca dyplomowa magisterska, autor mgr inż. Dariusz Aniszewski, promotor dr inż. Piotr Helt

How to make killer robots with Python and Raspberry Pi

Radomir Dopieralski

Robots in popular culture differ substantially from what is available today. You may see yourself in a role of a mad scientist constructing an army of robots to take over the world, but when you go to any website about building robots, you are presented with those... toys. They can't fly, don't have lasers and death rays, they ride around on wheels, and their main function is to avoid obstacles. Imagine that! You unleash your robot on the city to wreck havoc and destruction, and what it does when it faces a building or a police car? Meekly turns around and rides on its silly plastic wheels? No! The robots I build will trample the policemen and walk straight into the building, with no regard to their own safety. That's how it's done!

And no wheels. Wheels are easy – you basically only need an H-bridge and two electric motors with gear boxes. And a small battery. Connect the bridge to two GPIO pins, and you have full control. Pathetic.

Now, legs. Legs are interesting. First of all, your robot will need to be strong enough to carry its own weight, plus some extra strength for inertia. That means servomotors with enough torque, and a battery with low enough internal resistance to power them. If you want anything better than the silly chopstick walkers, or the ridiculous bipedals with huge feet, you will also need a lot of those servomotors. At least three per leg, to have full inverse kinematics. Multiply that with the number of legs, from four to eight, and you you get from 12 to 24 servomotors. Expensive and power-hungry, but that's the price of awesomeness!

You can improve the situation a little by initializing the servomotors in a sequence, not all at once, and by adding a large capacitor next to the battery – that will smoothen out any spikes in power consumption. Of course you won't connect all those servomotors to your central unit directly, it doesn't even have so many pins. You will need a servo controller, and you will need to send the commands to it through the serial interface or I²C, depending on what you get. You can even make your own out of an Arduino, especially if you need some custom functionality.

Now, suppose you have all the parts, and you start assembling it all together. It's easy to just glue everything together and solder all the wires. But then you realize, that you attached one of the legs up-side down, one of the servomotors is fried and you need a larger capacitor. So you have to break it all apart again. What a pain! Better to use gold pins, plugs, screws, and even velcro tape. This way you can easily modify your prototype and replace parts. Oh, and by the way, have some spares on hand. They *will* break. That's how the physical world works.

As for software, it's in equal parts programming and system administration. Remember that this is practically a mobile data center – multiple electronic devices communicating with each other. For instance, to use the serial interface you will need to edit `/boot/cmdline.txt` and `/etc/inittab` to disable the system console that normally runs there. And to use I²C, you will need to comment out the blacklisted modules from `/etc/modprobe.d/raspi-blacklist.conf`, and so on. Even if your program doesn't need any fancy settings or permissions, you will still need to make it run at system startup, or have some other way to start it.

I won't write much about the actual code here. After all, you are a programmer and it's what you want to solve by yourself. Just a few hints. Look for "inverse kinematics" – that's what you will need to translate the coordinates of the leg tips into actual servomotor angles. And the position of the robot's body into the positions of individual legs. Then, remember that animating a physical thing is very similar to animating computer graphics – just instead of writing pixels to the screen, you are sending commands to the servos. You are on your own with the rest.

Controlling your robot can also be a challenge. You might try to go fully autonomous, give it a lot of sensors, connected over ISP, I²C, 1Wire, TTL, USB or even directly to the pins as analog input. There is a wide range of sensors available, from proximity (both sonar and light-based) and motion sensors, through light, color, temperature, ambient noise, touch, acceleration, direction, GPS, to moisture. You can even have a camera and use OpenCV for image analysis! However, it's nice to have some kind of manual override. You can use a WiFi dongle and simply SSH to it. Or get one of those wireless game pads. Just remember that the Sony Sixaxis controller is not supported very well, especially if it's not the original one. You could even have your computer talk to your robot through a Bluetooth or IRDA or one of those transmitter-receiver sets for

Arduino. Whatever you choose, make sure it's always possible to just connect a good old monitor screen and keyboard to it, in case something goes very wrong and you lose access.

Finally, there is the equipment that your robot will carry. Of course, crawling around and trampling enemies is cool in itself, but you can make it even better. A speaker for making beeps or playing the `DESTROY.wav` file over and over (remember to get a small audio amplifier too, so that it can be heard). Blinking LEDs. A nerf rocket turret. Maybe even an Airsoft gun, so that you can take part in a Mech Warfare contest?

Last, but not least, remember about your own safety. Conquering the world is so much harder when you have no hands and you are bleeding from your eyes. Don't look into lasers, don't use power tools without glasses, don't play with high voltages, always be careful which end of the soldering iron you are grabbing. And remember that the LiPo batteries will erupt in flames if you so much as look funny at them. Seriously, search YouTube for videos of burning LiPos. You don't want that to happen in your secret lair. And never shoot anything at humans – you don't want them to become suspicious before your plans are fully prepared!

Now go forth and create robots! It's great fun. It's also addictive, expensive and takes all your free time – a perfect hobby.