The benefits of static typing without static typing in Python

One of the most popular complaints against Python is that its dynamic type system makes it easy to introduce bugs into your programs. As you probably know in statically typed languages (e.g. Java, C++, Rust) type of variable is checked at compile time. In dynamically typed languages (e.g. Python, Ruby) type of variables is interpreted at runtime. Proponents of statically typed languages argue that lack of type checking leads to bugs. If you have statically typed language and you make programming error of operating on incompatible types you get loud error from compiler. Your program is dead and will never go live. Take following “guess my secret number” game written in Rust for example.

```
extern crate rand;
use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    let secret = rand::thread_rng().gen_range(1, 101);
    println!("Please guess secret number");
    println!("Hint: secret number is {}", secret);
    println!("Please input your number");
    let mut guess = String::new();
    io::stdin()
        .read_line(&mut guess)
        .expect("failed to read");
    println!("your guess is: {}", guess);

    match guess.cmp(&secret) {
        Ordering::Less => println!("too small"),
        Ordering::Greater => println!("too big"),
        Ordering::Equal => println!("you win!")
    }
}
```

When you run above code sample Rust won’t compile, it will throw type error at you.

```
~/r/f/guess> cargo run
   Compiling guess v0.1.0 (file:///home/pawel/rusty/first_game/guess)
src/main.rs:16:21: 16:28 error: mismatched types:
 expected `&collections::string::String`,
    found `&_`
(expected struct `collections::string::String`,
    found integral variable) [E0308]
```

```
src/main.rs:16     match guess.cmp(&secret) {
                                ^~~~~~~
src/main.rs:16:21: 16:28 help: run `rustc --explain E0308` to see a detailed explanation
error: aborting due to previous error
Could not compile `guess`.
```

It may be a pain to see compile errors, but trust me, getting error here is good for you. You did something stupid you tried to compare string with int and decide which one is equal. This does not make any sense to the computer most of the time (nor does it make sense to human), so you should never be able to run program like this. Computer says “No!” and you have to cope with that.

Now try doing same idiotic thing in Python.

```
import random
# use six because input behaves differently in Python 3 and 2
import six

def main():
    secret = random.randint(1, 101)
    print("Guess secret number")
    print("Hint secret number is {}".format(secret))
    guess = six.moves.input("please input your number")
    print("Your guess is {}".format(guess))

    def compare(guess, secret):
        if guess == secret:
            print("you win")
        elif guess > secret:
            print("too big")
        elif guess < secret:
            print("too small")

    compare(guess, secret)

if __name__ == "__main__":
    main()
```

No pain here and you may even feel really productive. Your program may look harmless to novice programmer, but bug introduced here is pretty dangerous and surprisingly common. You are trying to compare string taken from standard input with integer returned by random.randint(), so you may end up with comparing “12” with 12.

In Python 2 and 3 doing “12” == 12 is allowed and does not break anything it just always evaluates to False. Python 2 also allows “greater than”, “smaller than” comparisons between incompatible types. If you do this kind of comparison between numeric and non-numeric type (“12” > 12) non-numeric type will always be greater. In contrast to Rust your Python script will happily compile and run without problems. This is bad because it will generate invalid output. This means that your user always looses the game no matter what he enters.

```
~/f/stack> python guess.py
secret number is 49
please input your number49
Your guess is 49
too big
```

Python 3 behaves in a different (better!) way. When doing greater than or smaller than comparisons it will explicitly complain that you are trying to compare incompatible types:

```
~/f/stack> python3 guess.py
secret number is 100
please input your number100
Your guess is 100
Traceback (most recent call last):
  File "guess.py", line 21, in <module>
    main()
  File "guess.py", line 18, in main
    compare(guess, secret)
  File "guess.py", line 13, in compare
    elif guess > secret:
  TypeError: unorderable types: str() > int()
```

This is much better - at least you don't get bad output, your program crashes early and lets you know you made a mistake. This means you can learn about the problem from your logs and not from users complaining that they cannot guess their numbers no matter how hard they try. It is still worse than Rust though, in Rust this would never launch or run, you'd get error immediately after trying to compile .

Also in Python 3 you can still check incompatible types for equality. So imagine you were lazy and wrote your function like this:

```
import random

def compare(guess, secret):
    if guess == secret:
        print("you win")
        return True
    else:
        print("try again loser!")
        return False
```

```
if __name__ == "__main__":
    secret = random.randint(1, 101)
    print("secret number is {}".format(secret))

    game_over = False
    while not game_over:
        guess = input("please input your number")
        print("Your guess is {}".format(guess))
        game_over = compare(guess, secret)
```

Now you have program that relies on equals comparison that will never return True. Your user is stuck in his game and cannot guess anything, he’s getting frustrated and angry and you have to fix it. Usually you don’t have this helpful print statement telling you what is actual secret number, and your program is long and often complicated. Not all users are good at communicating their needs so most of the time you have to figure out what exactly is the problem. Is the problem real or is it resulting from users cognitive limitations. Is he really guessing the right number? Maybe he’s just unlucky all the time? Maybe he enters something unusual? Finally you have to dig your way through your codebase and find this stupid trivial error you or someone else made. This is usually not productive and if you have done this more than once you start to get dislike Python for allowing people to make errors like this.

Static type checkers to the rescue

Luckily Python community is aware of limitations of dynamic typing and there are attempts to fix the problem.

One really cool attempt to fix this are type annotations. Type annotations allow you to specify type of function parameters and return values. They are described in PEP 0484. Sample syntax looks like this:

```
from typing import Iterator

def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a + b
```

This is valid Python, it will work completely fine in Python 3, will fail in Python 2. By default using type annotations syntax does nothing. Your program will still compile normally even if it contains invalid type operations. Let’s add type annotations to our game and see what happens.

```
import random

def compare(guess: int, secret: int) -> bool:
    if guess == secret:
        print("you win")
        return True
    else:
        print("try again loser!")
        return False

if __name__ == "__main__":
    secret = random.randint(1, 101)
    print("secret number is {}".format(secret))

    game_over = False
    while not game_over:
        guess = input("please input your number")
        print("Your guess is {}".format(guess))
        game_over = compare(guess, secret)
```

Now try to run it and wait. Surprise surprise… Nothing happens. This is by design. Python is dynamically typed and it will stay like this. Authors of this PEP dont want to change design of the language. They just want to make static type checks easier. So they propose to create static type checker built into the library. Something that could be used by users optionally. One library is gaining widespread support here, it is called mypy and it's author Jukka Lehtosalo is also one of the authors of PEP 0484.

MyPy acts same way as linter, it simply checks your program for type errors by looking at type annotations. So you pass your program to mypy and you get error saying that you are passing invalid type to your function call.

```
 ~/f/stack> mypy guess.py

guess.py:19: error: Argument 1 to "compare" has incompatible type "str"; expected "int"
```

If MyPy gets mature and is actually built into Standard Library we can get the best of both worlds. On the one hand people who like static languages can choose to use static type checkers. Those who prefer benefits of dynamic typing will stick to dynamic typing.

Probably this will still not be enough for Python enemies though. I'm pretty sure we're still going to read rants on Reddit complaining that Python is not enterprise ready yet because it's not statically typed. But I guess at least now every time you hear that you can show them PEP 0484 and tell them community is working on it.