Match case

The switch statement

As said before, Python has no switch statement. This means this code in PHP:

```php
switch ($language) {
    case 'French':
        $hello = 'Bonjour';
        break;
    case 'German':
        $hello = 'Hallo';
        break;
    case 'Italian':
        $hello = 'Ciao';
        break;
    case 'Spanish':
        $hello = 'Hola';
        break;
    default:
        $hello = 'Hello';
}
```

would be represented like this in Python:

```python
if language == 'French':
    hello = 'Bonjour'
elif language == 'German':
    hello = 'Hallo'
elif language == 'Italian':
    hello = 'Ciao'
elif language == 'Spanish':
    hello = 'Hola'
else:
    hello = 'Hello'
```

📌 An efficient and more elegant alternative in this example would be to just use an associative array in PHP or a dictionary in Python.

```python
hello_list = {
    'French': 'Bonjour',
    'German': 'Hallo',
    'Italian': 'Ciao',
    'Spanish': 'Hola',
}

hello = hello_list.get(language, 'Hello')
```

But this simple example is just to talk about switch statements and match cases.

Both solutions (the switch statement in PHP and the if/elif/else syntax in Python) have drawbacks. They're not so elegant and PHP's switch statement is not type-safe.

```php
$x = 12;

switch ($x) {
    case true:
        echo 'Yes';
        break;
    default:
        echo 'No';
}

// 'Yes'
```

The variable is compared with `==` instead of `===`, and `$x == true` returns `true`.

In both languages, a recent *match case* construct appeared (PHP 8 and Python 3.10). Python has a more evolved match case, but we will first see how similar they are.

Replacing switch with match

Let's take the previous switch example and use the match case instead.

```php
$hello = match ($language) {
    'French' => 'Bonjour',
    'German' => 'Hallo',
    'Italian' => 'Ciao',
    'Spanish' => 'Hola',
    default => 'Hello',
};
```

```python
match language:
    case 'French':
        hello = 'Bonjour'
    case 'German':
        hello = 'Hallo'
    case 'Italian':
        hello = 'Ciao'
    case 'Spanish':
        hello = 'Hola'
    case _:
        hello = 'Hello'
```

Two first differences can be spotted. First, Python's match case is not an expression, which means you cannot assign its result directly to a variable contrary to PHP. Second, the *default* case is represented by `case _`.

Both PHP and Python's match cases are type-safe.

Combined cases

```php
$hello = match ($language) {
    'French', 'Belgian French' => 'Bonjour',
    'German', 'Austrian German' => 'Hallo',
    'Italian' => 'Ciao',
    'Spanish' => 'Hola',
    default => 'Hello',
};
```

```python
match language:
    case 'French' | 'Belgian French':
        hello = 'Bonjour'
    case 'German' | 'Austrian German':
        hello = 'Hallo'
    case 'Italian':
        hello = 'Ciao'
    case 'Spanish':
        hello = 'Hola'
    case _:
        hello = 'Hello'
```

Unions are represented with a pipe (`|`) in Python, but a comma (`,`) in PHP.

Pattern matching

Now let's talk about what the match case in Python has more than the PHP version.

Python's match case allows pattern matching (it could actually **become a part of a future PHP version too**).

Let's consider a function to which we pass a few datetime-related arguments, and it tells us what parts we passed.

```python
def dissect_time(*dt):
    match dt:
        case [year]:
            print('We know the year')
        case [year, month]:
            print('We know the year and the month')
        case [year, month, day]:
            print('We know the year, the month and the day')
        case [year, month, day, *others]:
            print(
                'We know the year, the month,' \
                ' the day, and some time data'
            )

dissect_time(2020) # We know the year
dissect_time(2020, 4) # We know the year and the month
dissect_time(2020, 4, 22) # We know the year, the month and the day
# We know the year, the month, the day, and some time data
dissect_time(2020, 4, 22, 22, 30)
```

First, we pack the received parameters in a tuple. Then we match each pattern according to the number of arguments.

Patterns in patterns

We can go further. We can match only specific values while doing pattern matching.

Let's say we want a function to tell the time. The first argument should only be `AM` or `PM`, and we can give some parts of the time.

```python
def tell_time(*time):
    match time:
        case [('AM'|'PM'), hour]:
            print(f'It is {hour}')
        case [('AM'|'PM'), hour, minute]:
            print(f'It is {hour}:{minute}')
        case [('AM'|'PM'), hour, minute, second]:
            print(f'It is {hour}:{minute}:{second}')
        case _:
            print('Invalid time')

tell_time('AM', 8) # It is 8
tell_time('AM', 8, 22) # It is 8:22
tell_time('PM', 8, 22, 42) # It is 8:22:42
tell_time('FOO', 8, 22, 42) # Invalid time
tell_time(8, 22, 42) # Invalid time
```

As you can see, we can use the matching values between parentheses split with pipes (`|`).
If we give invalid parameters, we fallback on the default (`_`) case.

Now, what if we want to know if it's indeed `AM` or `PM` ? Right now, in our code there is no way to access the first matched parameter.

To do so, we need to use the `as` keyword.

```python
def tell_time(*time):
    match time:
        case [('AM'|'PM') as dayperiod, hour]:
            print(f'It is {hour} {dayperiod}')
        case [('AM'|'PM') as dayperiod, hour, minute]:
            print(f'It is {hour}:{minute} {dayperiod}')
        case [('AM'|'PM') as dayperiod, hour, minute, second]:
            print(f'It is {hour}:{minute}:{second} {dayperiod}')
        case _:
            print('Invalid time')

tell_time('AM', 8) # It is 8 AM
tell_time('AM', 8, 22) # It is 8:22 AM
tell_time('PM', 8, 22, 42) # It is 8:22:42 PM
tell_time('FOO', 8, 22, 42) # Invalid time
tell_time(8, 22, 42) # Invalid time
```

You can only specify one value and decompose differently your conditions.

```python
def tell_time(*time):
    match time:
        case ['AM', hour]:
            print(f'It is morning and it is {hour}')
        case ['AM', hour, minute]:
            print(f'It is morning and it is {hour}:{minute}')
        case ['AM', hour, minute, second]:
            print(f'It is morning and it is {hour}:{minute}:{second}')
        # Here we match 'PM' and any other parts
        case ['PM', *_]:
            print('It is the afternoon!')
        case _:
            print('Invalid time')

tell_time('AM', 8) # It is morning and it is 8
tell_time('AM', 8, 22) # It is morning and it is 8:22
tell_time('PM') # It is the afternoon!
tell_time('PM', 8, 22, 42) # It is the afternoon!
tell_time('FOO', 8, 22, 42) # Invalid time
tell_time(8, 22, 42) # Invalid time
```

Conditions

You can also add a condition within your pattern.

```python
def tell_time(*time):
    match time:
        # Here the line is split with `\` for
        # formatting purpose, but you can
        # obviously keep it in one line
        case [('AM'|'PM') as day_period, hour] \
        if hour >= 0 and hour <= 12:
            print(f'It is {hour} {day_period}')
        case _:
            print('Invalid time')

tell_time('AM', 8) # It is 8 AM
tell_time('PM', 8) # It is 8 PM
tell_time('AM', 22) # Invalid time
tell_time('PM', -3) # Invalid time
tell_time('AM') # Invalid time
tell_time('PM') # Invalid time
```

Object matching

Another interesting feature of match cases in Python is the possibility to match an object.

```python
class User:
    def __init__(self, email, name=None):
        self.email = email
        self.name = name


def describe_user(user):
   match user:
       case User(email='admin@admin.com', name='Admin'):
           print('admin')
       case User(email='michael@office.com', name='Michael'):
           print('boss')
       case User(email='dexter.morgan@email.com'):
           print('serial killer')
       case _:
           print('invalid user')


describe_user(User("admin@admin.com", "Admin")) # admin
describe_user(User("michael@office.com", "Michael")) # boss
describe_user(User(
    "dexter.morgan@email.com",
    "Dexter"
)) # serial killer
describe_user(User("dexter.morgan@email.com")) # serial killer
describe_user(User("some-unknown-user@email.com")) # invalid user
describe_user(12) # invalid user
```

This is the equivalent:

```python
class User:
    def __init__(self, email, name=None):
        self.email = email
        self.name = name


def describe_user(user):
    if (
        isinstance(user, User)
        and user.email == "admin@admin.com"
        and user.name == "Admin"
    ):
        print("admin")
    elif (
        isinstance(user, User)
        and user.email == "michael@office.com"
        and user.name == "Michael"
    ):
        print("boss")
    elif (
        isinstance(user, User)
        and user.email == "dexter.morgan@email.com"
    ):
        print("serial killer")
    else:
        print("invalid user")

describe_user(User("admin@admin.com", "Admin")) # admin
describe_user(User("michael@office.com", "Michael")) # boss
describe_user(User("dexter.morgan@email.com", "Dexter")) # serial killer
describe_user(User("dexter.morgan@email.com")) # serial killer
describe_user(User("some-unknown-user@email.com")) # invalid user
describe_user(12) # invalid user
```

Which is way more verbose and less readable. If you want to use positional arguments in your patterns you can check this **link**.