De re-module van Python

Een beknopte inventarisatie

Inhoud

1	**Inleiding**	**2**
2	**Componenten van de module `re`**	**3**
2.1	Classes	3
2.2	Functies	3
2.3	Modules	3
2.4	Flags	3
3	**Componenten van de class `Pattern`**	**4**
3.1	Instance-methodes	4
3.2	Overige attributen	4
4	**Componenten van de class `Match`**	**5**
4.1	Instance-methodes	5
4.2	Overige attributen	5
5	**Componenten van de class RegexFlag**	**6**
5.1	Properties	6
5.2	Instances (flags)	6
6	**Reguliere expressies, de taal zelf**	**7**
6.1	Metatekens	7
6.2	Escape-reeksen	8
6.3	Ankers	8
6.4	Tekenverzamelingen	8
6.5	Kwantoren	9
6.6	Alternatieven	10
6.7	Captures	10
6.8	Extra’s	10
7	**Uitgeleide**	**11**
Appendix A	**Vindt “vind alles” alles?**	**12**
Appendix B	**Klassenmodellen**	**13**
Appendix C	**Escape-reeksen**	**14**
Appendix D	**Referenties**	**16**

1 Inleiding

Het boek *De programmeursleerling* van Pieter Spronck [1] gaat over programmeren in Python. Grotendeels althans. Eén hoofdstuk, te weten hoofdstuk 25, behandelt een volkomen andere taal, die als concept weliswaar volledig op zichzelf staat, maar tegenwoordig in vele programmeertalen is ingebed. Waaronder ook in Python. Het is de taal der reguliere expressies.

Voor wie er niet mee bekend is, is het misschien wel even wennen. Het zit hem dan niet zozeer in de omvang van de taal, als wel in de totaal andere denkwijze die nodig is om er zelf in te kunnen programmeren. Het is in tegenstelling tot Python namelijk geen procedurele taal, maar een declaratieve. Je moet hierin niet beschrijven welke handelingen de computer moet verrichten, maar waar het resultaat aan dient te voldoen (net zoals dat voor bijvoorbeeld SQL geldt, al oogt laatstgenoemde wat mensvriendelijker). Ik kom daar later op terug.

Eerst laat ik zien hoe de reguliere expressies zijn ingebed in de Python-omgeving. De interactie tussen beide talen wordt via de vertrouwde procedurele mechanismen gerealiseerd. Dus met gebruikmaking van Python-statements en -functies, en aangezien Python een objectgeoriënteerde taal is, ook van classes en methodes.

Dit document is niet veel meer dan een oppervlakkige inventarisatie van alle interfaces tussen Python en reguliere expressies (hoofdstuk 2 t/m 5) en de verschillende taalelementen van reguliere expressies zelf (hoofdstuk 6). Ik heb daarbij aangegeven welke elementen in het boek van Spronck zijn behandeld en welke niet. Veel uitleg geef ik verder niet bij deze opsommingen. Ga waar mogelijk bij het boek te rade en anders bij de officiële Python-documentatie [2, 3]. Op mijn website [4] heb ik bovendien een kleine introductie staan die ik in 2009 voor mijn toenmalige collega's had geschreven (in de context van Perl en Ruby, wat maar heel weinig uitmaakt). De achtergronden van reguliere expressies worden belicht in Wikipedia [5].

Van sommige onderwerpen vind ik echter dat ik ze toch wel hier moet bespreken. Er zijn enkele faciliteiten waar Spronk naar mijn idee onterecht aan voorbij gaat. Die heb ik gemerkt met een naar links wijzende driehoek ◀. Daarnaast is er één voorziening die ik met ▶ als (zeker voor beginners) overbodige luxe heb aangemerkt.

Ik heb geprobeerd mij zo veel mogelijk aan de in of rond Python gangbare notaties te houden. Waar ik geen kans zag om daarmee goed uit te drukken wat ik bedoelde, heb ik teruggegrepen naar een BNF-dialect van eigen makelij dat ik Welsh Rarebit heb genoemd:

α ǀ β = α of β *(in de tabellen van paragraaf 4.1, 5.1, 5.2 en 6.7)*
α,··· = één of meer keer α gescheiden door een komma
α° = nul of één keer α
α,···° = nul of meer keer α gescheiden door een komma *(in de tabel van paragraaf 4.1)*
⁽, α⁾° = nul of één keer komma + spatie + α *(in de tabel van paragraaf 3.1)*

Al met al hoop ik dat dit geschrift enig houvast biedt aan degenen die hun weg proberen te vinden in de weelderige wereld van reguliere expressies binnen Python.

2 Componenten van de module re

2.1 CLASSES

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig
`re.Match`	`re.error`
`re.Pattern`	`re.RegexFlag`
	Ongedocumenteerd:
	`re.Scanner`

Spronk bestempelt de classes **`Match`** en **`Pattern`** niet expliciet als onderdelen van de module **`re`**, dus voor de volledigheid maak ik er hier toch nog maar even gewag van. De returnwaarde van de functie **`re.compile`** is een **`Pattern`**-object, terwijl de functies **`re.match`**, **`re.fullmatch`** en **`re.search`** een **`Match`**-object retourneren.

`Pattern` is nader beschreven in hoofdstuk 3, **`Match`** in hoofdstuk 4 en **`RegexFlag`** in hoofdstuk 5.

2.2 FUNCTIES

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig
re.compile(*pattern*, *flags*=0)	**re.escape**(*pattern*)
re.<u>findall</u>(*pattern*, *string*, *flags*=0)	**re.<u>fullmatch</u>**(*pattern*, *string*, *flags*=0)
re.<u>finditer</u>(*pattern*, *string*, *flags*=0)	**re.purge**()
re.<u>match</u>(*pattern*, *string*, *flags*=0)	◀ **re.<u>split</u>**(*pattern*, *string*, *maxsplit*=0, *flags*=0)
re.<u>search</u>(*pattern*, *string*, *flags*=0)	**re.<u>subn</u>**(*pattern*, *repl*, *string*, *count*=0, *flags*=0)
re.<u>sub</u>(*pattern*, *repl*, *string*, *count*=0, *flags*=0)	*Ongedocumenteerd:*
	re.template(*pattern*, *flags*=0)

De onderstreepte functies zijn bovendien beschikbaar als methode van **`Pattern`**-instances, zie paragraaf 3.1. Daar vertel ik er wat meer over.

2.3 MODULES

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig
	Ongedocumenteerd:
	`re.copyreg`
	`re.enum`
	`re.functools`

2.4 FLAGS

Instances van de class **`RegexFlag`**, zie hoofdstuk 5.

3 Componenten van de class `Pattern`

3.1 INSTANCE-METHODES

Door Spronck behandeld		Bovendien in Python 3.12 aanwezig	
pattern.**findall**(*string*[, *pos*[, *endpos*]])	list	*pattern*.**fullmatch**(*string*[, *pos*[, *endpos*]])	Match
pattern.**finditer**(*string*[, *pos*[, *endpos*]])	*iterator*	◀ *pattern*.**split**(*string*, *maxsplit=0*)	list
pattern.**match**(*string*[, *pos*[, *endpos*]])	Match	*pattern*.**subn**(*repl*, *string*, *count=0*)	tuple
pattern.**search**(*string*[, *pos*[, *endpos*]])	Match		
pattern.**sub**(*repl*, *string*, *count=0*)	str		

Deze methodes zijn allemaal ook beschikbaar als functie in de module **re**, zie paragraaf 2.2. De methodes die worden geacht een **Match**-object aan te leveren, retourneren **None** indien er geen match is gevonden.

Helaas strooit Spronck wat te veel met het woord ‘patroon’ wanneer hij op de uitvoer van de **findall**-functie of -methode doelt. Die uitvoer is een list van substrings die aan een zeker **Pattern**-object (kortom, aan een patroon) voldoen of een list van tuples van zulke substrings (zie Appendix A). Het zou beter zijn geweest de listelementen niet met ‘patronen’ maar met ‘vondsten’ of ‘vondstobjecten’ (Engels: *finds* of *find objects*) aan te duiden.

De iterator die door **finditer** wordt geleverd is een **callable_iterator**-object waarmee je via het **for**-statement over een collectie van **Match**-objecten kunt itereren.

Het argument *repl* in **sub** en **subn** biedt een vervangende string aan, maar mag ook de naam van een functie zijn die een string retourneert:

- Is *repl* een uitdrukking die bij evaluatie een string oplevert, dan wordt deze string voor elke vondst die aan *pattern* voldoet in de plaats gesteld en toegevoegd aan vorige, zodat de uiteindelijke string die geretourneerd wordt een keten van substituties vormt.
- Is *repl* een uitdrukking die bij evaluatie een *callable object*[1] oplevert, dan wordt dat object bij elke match die door **finditer** zou worden gevonden aangeroepen met *repl*(*match*), waarbij *match* dat aangetroffen **Match**-object is. De callable moet een stringwaarde retourneren. De resultaten van alle successievelijke aanroepen worden aaneengeregen tot één lange string die uiteindelijk door **sub** of **subn** wordt geretourneerd.

De **subn**-methode voegt bovendien het aantal matches toe, zodat er een tuple van twee elementen ontstaat, een string en een integerwaarde.

De methode *pattern*.**split** (en daarmee ook de functie **re.split**) is een welkome aanvulling op de methode *string*.**split** wanneer je een string wilt opsplitsen op basis van een scheiderpatroon dat zich niet in de ingebouwde stringmethode laat omschrijven. Ik heb vaak genoeg reguliere expressies voor zoiets nodig gehad.

3.2 OVERIGE ATTRIBUTEN

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig	
	pattern.**flags**	int
	pattern.**groups**	int
	pattern.**groupindex**	dict
	pattern.**pattern**	str

[1] Denk aan een functie of methode.

4 Componenten van de class `Match`

4.1 INSTANCE-METHODES

Door Spronck behandeld		Bovendien in Python 3.12 aanwezig	
match.**end**(*group=0*, /)	int	◀ *match*.**__getitem__**(*key*, /)	str
match.**group**(*group*,…°)	str ǀ tuple	*match*.**expand**(*string*, *maxsplit=0*, *flags=0*)	str
match.**groups**(*default=0*)	tuple	*match*.**groupdict**(*default=0*)	dict
match.**start**(*group=0*, /)	int	*match*.**span**(*group=0*, /)	tuple

Tegen mijn voornemen in om geen dunders[2] te noemen, wil ik hier toch melding maken van *match*.**__getitem__**. Niet om je te bewegen deze methode direct aan te roepen, maar vanwege het neveneffect. Het stelt je in staat om de aanroepen *match*.**group(1)** en *match*.**group("A")** in te korten tot *match*[**1**] respectievelijk *match*[**"A"**]. Wanneer je de match in zijn geheel wilt ophalen, kan dat met zowel *match*.**group()** als *match*.**group(0)**, maar de enige korte vorm die daarbij is toegestaan, is *match*[**0**]. Slices zoals *match*[**2:4**] kunnen niet worden toegepast. Die zullen altijd de foutmelding “IndexError: no such group” geven.

4.2 OVERIGE ATTRIBUTEN

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig	
	match.**endpos**	int
	match.**lastgroup**	str
	match.**lastindex**	int
	match.**pos**	int
	match.**re**	Pattern
	match.**string**	str

[2] ‘Dunder’ staat voor ‘**d**ouble **under**score’. Daarmee kan **__getitem__** worden uitgesproken als ‘dunder get-item dunder’ of kortweg ‘dunder get-item’. Vervolgens is het woord in het spraakgebruik tot een zelfstandig naamwoord geëvolueerd en kun je zeggen dat **__getitem__** een dunder is.

5 Componenten van de class RegexFlag

5.1 PROPERTIES

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig

	Ongedocumenteerd:
	`RegexFlag.A` | `RegexFlag.ASCII`
	`RegexFlag.DEBUG`
	`RegexFlag.I` | `RegexFlag.IGNORE`
	`RegexFlag.L` | `RegexFlag.LOCALE`
	`RegexFlag.M` | `RegexFlag.MULTILINE`
	`RegexFlag.NOFLAG`
	`RegexFlag.S` | `RegexFlag.DOTALL`
	`RegexFlag.U` | `RegexFlag.UNICODE`
	`RegexFlag.T` | `RegexFlag.TEMPLATE`
	`RegexFlag.X` | `RegexFlag.VERBOSE`

De class **`RegexFlag`** is een instance van **`enum.EnumType`** en een subclass van **`IntFlag`**. Alle hier genoemde *properties* (methodes vermomd als attributen — Spronck besteedt geen aandacht aan dit fenomeen) verwijzen naar ingebouwde **`RegexFlag`**-instances, waarover in de volgende paragraaf meer.

Het klassenmodel tussen **`IntFlag`** en **`object`** is behoorlijk complex. Wie zich daar werkelijk eens in wil verdiepen (voor het kunnen toepassen van reguliere expressies volkomen overbodig), kan een eerste indruk opdoen in Appendix B. Een bijzondere eigenschap van **`RegexFlag`** is, dat het niet alleen een class van haar instances is, maar bovendien een verzameling ervan. Dat trekje blijkt via een aantal tussengelegen classes geërfd van de class **`Enum`**.

5.2 INSTANCES (FLAGS)

Door Spronck behandeld		Bovendien in Python 3.12 aanwezig	
`re.I` | `re.IGNORECASE`	2	`re.A` | `re.ASCII`	256
`re.M` | `re.MULTILINE`	8	`re.DEBUG`	128
`re.S` | `re.DOTALL`	16	`re.L` | `re.LOCALE`	4
		`re.NOFLAG`	0
		`re.U` | `re.UNICODE`	32
		`re.X` | `re.VERBOSE`	64
		Ongedocumenteerd:	
		`re.T` | `re.TEMPLATE`	1

De opgesomde modulevariabelen verwijzen naar de **`RegexFlag`**-instances die in de vorige paragraaf ook al als properties van de class **`RegexFlag`** werden genoemd. Python gebruikt zelf steeds de lange naam om zo'n 'flag' object aan te duiden:

```
>>> print(re.ASCII, re.A, re.A.name, re.A.value, int(re.A), sep=", ")
re.ASCII, re.ASCII, ASCII, 256, 256
```

Achter ieder object gaat een **`int`**-waarde schuil zoals aangegeven in de tabel.

Vermoei je maar liever niet met wat zich onder water allemaal afspeelt, en pas simpelweg de uitdrukkingen in de linker kolom toe wanneer je enige fine-tuning wenst (**`re.X`** in de rechter kolom is me trouwens ook wel eens van pas gekomen).

6 Reguliere expressies, de taal zelf

Wanneer je uit de tekst “Monty Python’s Flying Circus” de substring “Python’s Fly” zou willen selecteren, zou je daar een algoritme voor kunnen schrijven dat ongeveer deze lijn volgt:

- Zoek van links naar rechts het eerste voorkomen van de letter `P` en onthoud de index daarvan.
- Zoek van rechts naar links het laatste voorkomen van de letter `y` en onthoud de index daarvan.
- Selecteer een substring vanaf de eerste index t/m de tweede. Hebbes.

Python is een procedurele taal en beschikt over een aantal aardige stringmethodes, dus dat moet wel lukken. Interessant om dat eens uit te proberen en de hoeveelheid code te vergelijken met de declaratieve aanpak die nu volgt:

- Vind een substring die voldoet aan het patroon:
  de letter `P`; zo veel mogelijk willekeurige tekens; de letter `y`.

Gemakkelijker gezegd dan gedaan, maar het leuke is natuurlijk dat de programmeur alleen maar gemakkelijk hoeft te praten en het systeem maar moet uitzoeken hoe het die selectie voor elkaar te krijgt. Wat het gemakkelijk praten betreft, de wens kan nog aanmerkelijk bondiger worden geformuleerd:

- Vind een substring die voldoet aan het patroon:
  `P.*y`.

Het sterretje is een postfix-operator met de betekenis ‘zoveel mogelijk’ en de punt representeert een willekeurig teken. Deze uiterst compacte notatie wordt *reguliere expressie* genoemd.

Natuurlijk moeten er in echte Python-code nog een paar extra formaliteiten worden vervuld. Onderstaand programmaatje laat zowel de kern als alle plichtplegingen eromheen zien, en is in deze form volledig functioneel.

```
import re
print( re.search( "P.*y", "Monty Python's Flying Circus" )[0] )
```

De uitvoer van dit programma is: **`Python’s Fly`**.

6.1 METATEKENS

Een reguliere expressie is een keten van tekens die samen een bepaald patroon beschrijven. Daarbij representeren de meeste tekens zichzelf, maar sommige hebben een speciale betekenis op een ander niveau. Vandaar dat men van metatekens (Engels: *metacharacters*) spreekt. Dit zijn ze:

```
. ^ $ * + ? { } [ ] ( ) | \
```

Wil je dat zo’n metateken zichzelf voorstelt, dan moet je deze ‘escapen’ door er een backslash voor te zetten. Zo is `.` een ‘joker’ (Engels: *wildcard*) met de betekenis ‘een willekeurig teken’[3] en geeft alleen de combinatie `\.` aan dat letterlijk het puntteken `.` wordt bedoeld. Een letterlijk als zodanig op te vatten backslash moet dus worden genoteerd als `\\`. (Die laatste punt hoort er niet meer bij, die sluit alleen maar de vorige zin af.)

[3] Met één uitzondering: het *newline*-teken (ASCII-volgnummer 10, veelal geschreven als `\n` en ook wel *line feed* genoemd), matcht niet met `.`! Met de `re.S`-flag kan echter worden aangegeven dat ook de regelovergang als een willekeurig teken geldt.

6.2 ESCAPE-REEKSEN

Een zoekpatroon wordt in eerste instantie in de vorm van een string beschreven. Voordat deze werkzaam kan worden, moet de string eerst — impliciet of expliciet — worden omgezet naar een `Pattern`-object. In het stringstadium kunnen in beginsel de gebruikelijke escape-reeksen (Engels: *escape sequences*) voor strings worden toegepast, zoals `\n`, `\t` en andere (zie de hoofdstukken 3 en 10 van Sproncks boek).[4] Ten behoeve van de patroonbeschrijving zijn er echter een aantal escape-reeksen toegevoegd:

Door Spronck behandeld		Bovendien in Python 3.12 aanwezig
`\b` en de tegenhanger `\B`	(afkorting van ***b**oundary*)	`\A`
`\d` en de tegenhanger `\D`	(afkorting van ***d**igit*)	`\Z`
`\s` en de tegenhanger `\S`	(afkorting van *white**s**pace*)	
`\w` en de tegenhanger `\W`	(afkorting van ***w**ord*)	

Behalve de ankers (zie volgende paragraaf) representeren ze een teken uit een bepaalde tekenverzameling of juist daarbuiten. Het gaat dan respectievelijk om cijfers, witruimte en 'woordtekens' (onder die laatste categorie worden letters, cijfers en het onderstrepingsteken oftewel *underscore* geschaard).

Hoewel Spronck veel aandacht aan de escape-reeksen besteedt, blijft het lastig om het geheel van mogelijkheden en onmogelijkheden te overzien. In Appendix C heb ik het daarom allemaal maar eens in kaart gebracht.

6.3 ANKERS

Onderstaande uitdrukkingen fungeren als anker (Engels: *anchor*):

Door Spronck behandeld	Bovendien in Python 3.12 aanwezig
`^` en `$`	`\A` en `\Z`
`\b` en `\B`	

Dat houdt in, dat ze niet matchen met bepaalde tekens, maar met bepaalde plaatsen binnen een string, namelijk waar zich een zeker soort overgang (stringgrens, woordgrens, nieuwe regel) bevindt. Zo'n overgang neemt geen ruimte in.

6.4 TEKENVERZAMELINGEN

Een tekenverzameling (Engels: *character set*) wordt aangegeven met `[`*sequence*`]`, waarbij *sequence* bestaat uit een opsomming van tekens die tot die verzameling behoren. Ook mogen bereiken van tekens worden opgegeven. In `[a-zA-Z0-9_]` wordt gesteld dat alle letters `a` t/m `z` en `A` t/m `Z`, alle cijfers `0` t/m `9` en underscores tot de tekenverzameling behoren (wat in dit geval overeenkomt met de escape-reeks `\w`). Er wordt op die manier gevraagd om een match met één van de tekens uit de verzameling.

Je kunt ook een negatieve tekenverzameling opvoeren: `[^`*sequence*`]`. Met `[^a-zA-Z0-9_]` vraag je dus om een match met één teken dat juist geen letter, cijfer of underscore is (wat ook met `\W` kan worden gedaan).

[4] De minder bekende zijn: `\a` (alarm, ASCII-teken 7), `\b` (backspace, ASCII 8) `\f` (form feed, ASCII 8), `\r` (carriage return, ASCII 13) en `\v` (vertical tab, ASCII 11). Behalve `\r` gebruik ik deze zelden of nooit.

De volgende tekens zijn binnen een tekenverzameling géén metateken zoals opgesomd in paragraaf 6.1 en worden dus letterlijk als punt, dollar, asterisk enzovoort opgevat:

```
. $ * + ? { } [ ( ) |
```

Ze mogen worden geëscapet, maar het hoeft niet. Van de volgende tekens hebben de eerste twee meestal en de laatste twee altijd een speciale betekenis:

```
^ - ] \
```

Om een letterlijke betekenis aan `]` en `\` toe te kennen moet er altijd een backslash voor worden gezet. Bij de andere twee zijn daar andere mogelijkheden toe. Een `^` wordt al letterlijk genomen wanneer dit teken ergens anders binnen de tekenset staat dan aan het begin ervan. Het koppelteken verliest zijn speciale betekenis juist als deze wèl aan het begin van de tekenset staat, of aan het eind ervan. In Appendix C zijn de escape-reeksen binnen tekensets geïnventariseerd.

6.5 KWANTOREN

Met een kwantor (Engels: *quantifier*) kan worden aangegeven hoe vaak een teken moet voorkomen:

Door Spronck behandeld		Bovendien in Python 3.12 aanwezig
{*n*,*m*}	minimaal *n* keer, maximaal *m* keer	
{*n*,}	*n* of meer keer	
{*n*}	precies *n* keer	
*****	= **{0,}**	
+	= **{1,}**	
?	= **{0,1}**	

Kwantoren hebben een hoge prioriteit: ze binden aan het onmiddellijk voorafgaande teken. Om meer dat één teken tegelijk te herhalen, moet je ze groeperen met ronde haakjes. Waar **ABC*** een patroon van één **A**, één **B** plus een onbepaald aantal **C**’s beschrijft, specificeert **(ABC)*** een onbepaald aantal **ABC**-combinaties.

De kwantoren kunnen nog met een extra modificator (Engels: *modifier*) worden uitgerust om nadere richtlijnen te geven over de wijze van uitvoering.[5] Duiden we een willekeurige kwantor uit bovenstaande tabel aan met ℚ, dan zijn de volgende combinaties mogelijk:

Door Spronck behandeld		Bovendien in Python 3.12 aanwezig	
ℚ	gulzig (Engels: *greedy*)	◀ **ℚ?**	lui (Engels: *non-greedy*, *lazy*)
		ℚ+	bezitterig (Engels: *possessive*)

De luie aanpak is echt wel wat aandacht waard. In de string “**Monty Python's Flying Circus**” komt het gretige patroon **P.*y** tot de match “**Python's Fly**”, terwijl het ingetogen **P.*?y** al tevreden is met de minimale match “**Py**”. Het ontbreken van een manier om een match zo klein mogelijk te houden zou werkelijk een serieuze handicap zijn, is mijn ondervinding.[6]

[5] Beschouw de kwantor en de eventuele modificator ervan maar als één ondeelbare operator.

[6] In dit geval zouden we kunnen uitwijken naar het patroon **P[^y]*y**, oftewel: er mogen zich tussen de gezochte **P** en de gezochte **y** geen andere **y**’s bevinden. Een ander alternatief is **P(?:(?!y).)*y**, met een geneste constructie van twee in paragraaf 6.8 vermelde taalcomponenten. Maar wat mij betreft: leve de ***?**-operator!

6.6 ALTERNATIEVEN

De operator **|** (uit te spreken als “of” of “or”) scheidt alternatieve patronen van elkaar. Deze operator heeft een lage prioriteit. Wil je delen van een patroon als plaatselijke alternatieven opvoeren, dan moet je dit stelsel van alternatieven groeperen door middel van ronde haakjes rondom dat stelsel. Het patroon `Hä|ae?ndel` probeert te matchen met “`Hä`”, `andel` of “`aendel`”. Dat is waarschijnlijk niet de bedoeling. `H(ä|ae?)ndel` zal dan wel het gewenste resultaat geven.

6.7 CAPTURES

Reguliere expressies bieden voorzieningen om plaatselijke matches met een deel van het patroon op te vangen voor later gebruik. Met het tussen haakjes zetten van een patroondeel ontstaat namelijk een genummerde groep waarin de substring wordt opgeslagen die aan het betreffende deel voldoet. De nummering begint bij **1** en loopt vervolgens van links naar rechts met de openingshaakjes op (tot maximaal **99**). Goed om te weten als je deelpatronen gaat nesten.

<table>
<tr><th colspan="2">Door Spronck behandeld</th><th colspan="2">Bovendien in Python 3.12 aanwezig</th></tr>
<tr><td>(subpattern)</td><td>genummerd deelpatroon</td><td></td><td></td></tr>
<tr><td>(?P<<i>name</i>><i>subpattern</i>)</td><td>benoemd deelpatroon ▶</td><td>(?P=<i>name</i>)</td><td>verwijzing naar groep binnen patroon</td></tr>
<tr><td>\n | \nn</td><td>verwijzing naar groep binnen patroon</td><td>\n | \nn</td><td>verwijzing naar groep in vervanging</td></tr>
<tr><td>\g<<i>n</i>> | \g<<i>nn</i>></td><td>verwijzing naar groep in vervanging</td><td>\g<<i>name</i>></td><td>verwijzing naar groep in vervanging</td></tr>
</table>

Naast numerieke identificatie bestaat tevens de mogelijkheid groepen betekenisvolle namen toe te kennen. Ik heb daar sinds 2000, toen ik met reguliere expressies begon, eigenlijk nooit behoefte aan gevoeld. Er staat namelijk ook een nadeel tegenover: de reguliere expressie wordt er een graadje ‘hariger’ van. Benoemde groepen blijven te benaderen via hun groepsnummer.

Eigenlijk vallen de **(?P…)**-constructies historisch gezien onder de noemer ‘latere extraatjes’. Meer daarvan in de volgende paragraaf.

6.8 EXTRA’S

Op één constructie na zul je waarschijnlijk maar weinig gebruik maken van onderstaande uitdrukkingen.

<table>
<tr><th>Door Spronck behandeld</th><th colspan="2">Bovendien in Python 3.12 aanwezig</th></tr>
<tr><td></td><td>(?#comment)</td><td></td></tr>
<tr><td></td><td>◀ (?:subpattern)</td><td></td></tr>
<tr><td></td><td>(?=subpattern) en (?!subpattern)</td><td>lookahead</td></tr>
<tr><td></td><td>(?<=subpattern) en (?<!subpattern)</td><td>lookbehind</td></tr>
<tr><td></td><td>(?>subpattern)</td><td></td></tr>
<tr><td></td><td>(?(cond)yesPat|noPat) en (?(cond)yesPat)</td><td></td></tr>
<tr><td></td><td>(?aiLmsux)</td><td>global flags</td></tr>
<tr><td></td><td>(?aiLmsux-imsx:subpattern)</td><td>local flags</td></tr>
</table>

Die uitzondering is **(?:***subpattern***)**. Die groepeert wel op dezelfde manier als **(***subpattern***)**, maar slaat geen captures op. Dat reduceert niet alleen het aantal genummerde groepen (fijn voor de overzichtelijkheid), maar komt ook de performance ten goede. En soms zitten die groepen zelfs ronduit in de weg, zie Appendix A. Ik prefereer daarom, terugkerend naar het voorbeeld in paragraaf 6.6, `H(?:ä|ae?)ndel` wanneer ik niet geïnteresseerd ben in aparte opvang van die **ä**, **a** of **ae**.

7 Uitgeleide

De compactheid van reguliere expressies heeft wel een ergonomische keerzijde. Wanneer ze meer dan zo'n twintig à dertig posities lang zijn, neemt de leesbaarheid nogal af. Gebezigde kwalificaties zoals 'harige code' en 'Snoopy swearing' zijn niet complimenteus bedoeld. Na wat training treedt heus wel gewenning op, maar de informatiedichtheid en het gebrek aan visuele redundantie in reguliere expressies is zeker niet optimaal op vertering door de menselijke lezer toegesneden. Dreigt het werkelijk te gek te worden, dan zijn er wel manieren om de code beter behapbaar te maken, onder meer door gebruik te maken van de *extended syntax*. Zie bijvoorbeeld [6]. In mijn eigen praktijk heb ik daar maar spaarzaam mijn toevlucht toe hoeven nemen.

In economisch opzicht hangt er ook een prijskaartje aan het gebruik van reguliere expressies. De verwerking ervan is doorgaans een orde van grootte trager dan wanneer je de beoogde functionaliteit zelf uitprogrammeert. Dat doet er echter alleen toe als performance een belangrijke factor is.

Het voordeel van reguliere expressies is, dat het zulk krachtig gereedschap vormt. Ik moet er niet aan denken dat ik overal waar ik ze heb toegepast, zelf algoritmes had moeten opstellen. Maar ondanks die kracht, er zijn grenzen. Niet alle stringanalyses kunnen met louter reguliere expressies worden uitgevoerd.

Zoals wel vaker bij populaire talen voorkomt, zijn er ook van reguliere expressies verschillende dialecten in omloop. Gelukkig bestaat er een de facto standaard in de vorm van *Perl Compatible Regular Expressions* (PCRE), waar veel implementaties (afgezien van eventuele eigen toevoegingen) aan voldoen. Daaronder bevindt zich ook Python, dus wat je hier leert, kun je (buiten enkele specifieke Python-uitbreidingen) in veel andere programmeertalen en tools net zo goed toepassen. Daarnaast bestaat er zoiets als *POSIX extended regular expressions*, eveneens op veel plekken ondersteund. Daar doet Python echter niet aan mee, en een groot gemis vind ik dat niet.

Ik heb in dit document uiteindelijk veel meer zaken toegelicht dan ik aanvankelijk van plan was. Daarmee is dit werk misschien wat onevenwichtig geworden. Bezie het dan liever als een half vol glas dan als half leeg. Mijn ambitie ging nu eenmaal niet verder dan het rijke materiaal dat in Python verborgen zit een beetje overzichtelijk uit te stallen, zodat de lezer een idee krijgt waar en waarnaar hij op zoek moet om meer over de mogelijkheden van reguliere expressies te weten te komen.

Appendix A Vindt "vind alles" alles?

Vertellen de functie en de methode **findall** ons werkelijk alles? Nee. Je zult moeten kiezen:

- Wanneer de reguliere expressie geen captures bevat, produceert **findall** een list van gevonden substrings.
- Wanneer de reguliere expressie wèl een of meer captures bevat, produceert **findall** een list van al dan niet gebundelde captures. Was er maar één capture, dan komt die als string in de list terecht. Waren er meer, dan worden de captures in volgorde van groepsnummer als string in een tuple opgenomen, en dat tuple op zijn beurt in de list. Heb er erg in, dat groep **1** binnen het tuple index **0** heeft! (Dat geldt ook voor *match*.**groups()** in paragraaf 4.1.)

Je krijgt dus òf alle voorkomens van substrings die aan het totale patroon voldoen, òf alleen de voorkomens van reeksen groepen. Er zijn een aantal technieken om toch van twee walletjes te eten. Om te beginnen kun je overbodige captures neutraliseren door de **(?:** ...**)** notatie toe te passen (zie paragraaf 6.8). Ben je niettemin werkelijk in een aantal captures geïnteresseerd, dan kun je van het gehele patroon eveneens een capture maken. Dat wordt dan uiteraard vanzelf de eerste.

Voor de aardigheid heb ik zelf maar een **findfully**-functie in elkaar geknutseld die de beperking van **findall** doorbreekt:

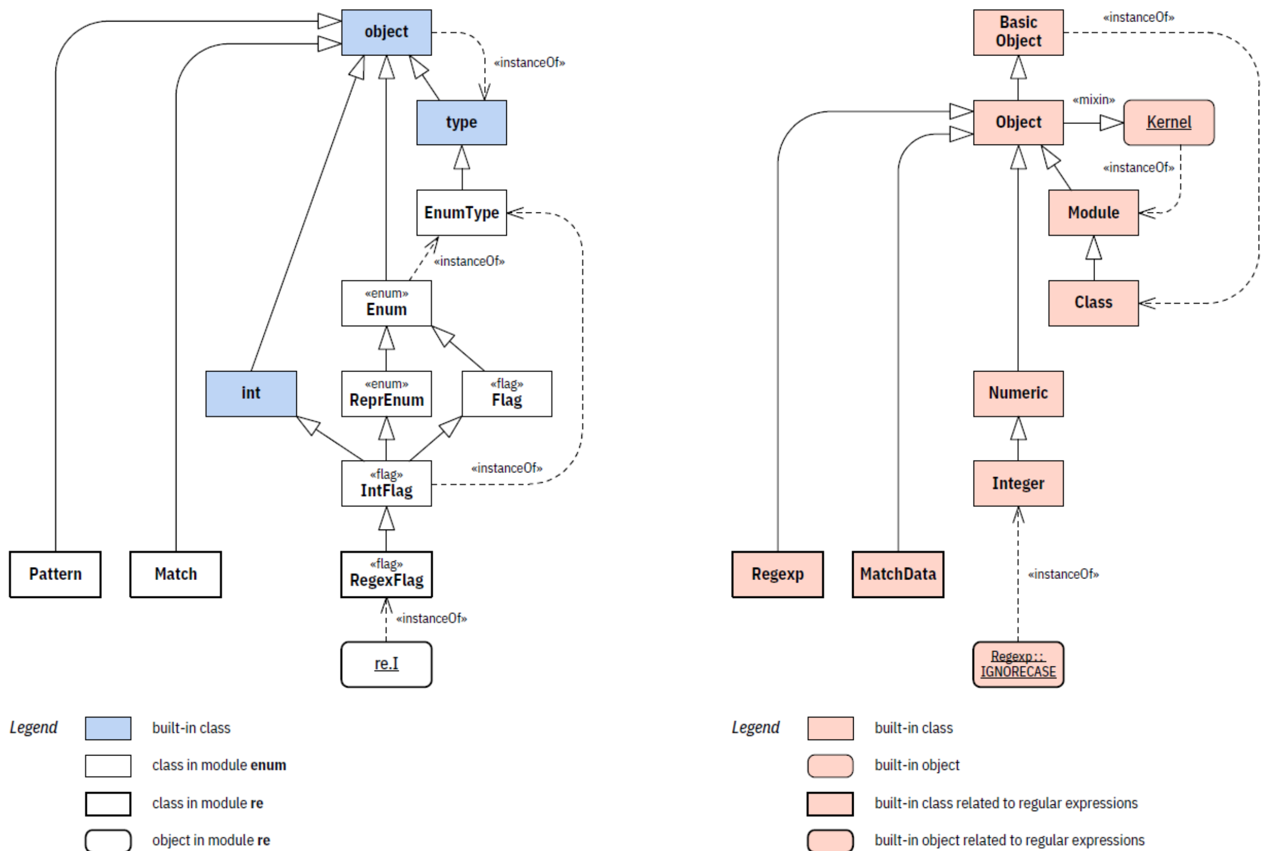
```
import re

def findfully(pat, text):
    matchitor = pat.finditer(text) if isinstance(pat, re.Pattern) else re.finditer(pat, text)
    return [(match[0],) + tuple([group for group in match.groups() if group]) for match in matchitor]

text = """\
Waar hebben Python-programmeurs het meest behoefte aan? Wat voor programmeurs?"""
#----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+--

regex0 = r"W.*?\?"
regex1 = r"W.*?(?:(.)\1.*?)+\?"
regex2 = r"W.*?(.)\1.*?(.)\2.*?(?:(.)\3.*?(.)\4.*?(.)\5.*?)?\?"
regex3 = r"(W.*?(.)\2.*?(.)\3.*?(?:(.)\4.*?(.)\5.*?(.)\6.*?)?\?)"
pat0   = re.compile(regex0)
print( "-" * 60 )

print(pat0.findall(text))
print(findfully(pat0, text))
print( "-" * 60 )

print(re.findall(regex1, text))
print(findfully(regex1, text))
print( "-" * 60 )

print(re.findall(regex2, text))
print(findfully(regex2, text))
print( "-" * 60 )

print(re.findall(regex3, text))
print(findfully(regex3, text))
print( "-" * 60 )
```

Deze functie levert voor iedere vondst van het gezochte patroon een tuple. Binnen zo'n tuple staat de gehele vondst op index **0**, groep **1** op index **1**, groep **2** op index **2**, enzovoort. Mooi. In **regex3** heb ik echter de hierboven beschreven truc toegepast: de hele reguliere expressie staat tussen haakjes en vormt dus groep **1**. Dat maakt **findfully** natuurlijk nogal overbodig. Maar het was leuk er even mee te spelen.

Appendix B Klassenmodellen

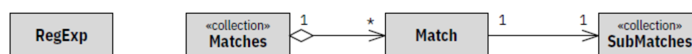
Nieuwsgierig geworden wat er achter **`re.RegexFlag`** schuilgaat, heb ik enige peilingen verricht en de resultaten daarvan opgetekend in het linker klassendiagram hieronder. De twee classes **`re.Pattern`** en **`re.Match`** zijn heel overzichtelijk: rechtstreekse specialisaties van de ingebouwde class **`object`**. Voor het banistieke deel is echter een uitgebreid ambtelijk apparaat opgetuigd. Hierin is **`re.RegexFlag`** een instance van de metaclass **`enum.EnumType`** en via meervoudige overerving aan tal van andere classes verbonden. Opvallend is ook de wijze waarop Python sommige classes aanduidt: het spreekt niet van `<class 'Enum'>` maar van `<enum 'Enum'>`, en evenzo van `<flag 'RegexFlag'>`. Niet iets om je druk over te maken, want het geheel doet geruisloos zijn heilzame werk.

In onderstaande diagrammen heb ik de «instanceOf»-afhankelijkheid van classes weggelaten indien deze gelijk is aan de «instanceOf»-afhankelijkheid van haar enige superclass.



Na van deze bevindingen te zijn bekomen, heb ik ook nog maar even nagetrokken hoe het bij Ruby in elkaar steekt. Dat levert het schema aan de rechterkant op. Alle reguliere-expressie-logica is hier in de taal ingebouwd. Vlaggen zoals **`Regexp::IGNORECASE`** zijn class-constanten met een alledaagse **`Integer`**-waarde.

VBScript is toch duidelijk anders ingericht, besef ik maar weer eens. Vier classes, waarvan twee een zogeheten *collection* zijn. Een model dat goed past in de opzet van VBScript.



Er wordt niet met vlaggen gewerkt, maar met *properties* zoals *oRegExp*.**`IgnoreCase`**, die op **`True`** kunnen worden gezet.

Appendix C Escape-reeksen

In onderstaande tabel is aangegeven hoe Python in verschillende situaties omspringt met de combinatie van een backslash (ASCII-teken 92) en een in de kolom **Char** genoemd teken uit de ASCII-reeks 32 t/m 127. In sommige gevallen maken vervolgtekens ook nog deel uit van wat als één geheel een specifieke betekenis krijgt.

Er worden vier situaties onderscheiden waarin ook andere varianten zijn te vangen:

- **Quot** — een gewone *string literal* tussen aanhalingstekens: (`"Hello!\n"`).
- **Pat** — een patroon dat via een *raw string* is gespecificeerd: (`r"Hello!\n"`).
- **Set** — de specificatie van een tekenset binnen een raw string: (`r"`…`[aeiou\n]`…`"`).
- **Repl** — een vervanging in de vorm van een raw string: (`r"Hello, \g<1>\n!"`).

Char	Quot	Pat	Set	Repl	Char	Quot	Pat	Set	Repl	Char	Quot	Pat	Set	Repl
SP		↘	↘		@		↘	↘		`		↘	↘	
!		↘	↘		A		⚓	⛔	⛔	a	▶	▶	▶	▶
"	↘	↘	↘		B		⚓	⛔	⛔	b	▶	⚓	▶	▶
#		↘	↘		C		⛔	⛔	⛔	c		⛔	⛔	⛔
$		↘	↘		D		▶	▶	⛔	d		▶	▶	⛔
%		↘	↘		E		⛔	⛔	⛔	e		⛔	⛔	⛔
&		↘	↘		F		⛔	⛔	⛔	f	▶	▶	▶	▶
'	↘	↘	↘		G		⛔	⛔	⛔	g		⛔	⛔	◁
(		↘	↘		H		⛔	⛔	⛔	h		⛔	⛔	⛔
)		↘	↘		I		⛔	⛔	⛔	i		⛔	⛔	⛔
*		↘	↘		J		⛔	⛔	⛔	j		⛔	⛔	⛔
+		↘	↘		K		⛔	⛔	⛔	k		⛔	⛔	⛔
,		↘	↘		L		⛔	⛔	⛔	l		⛔	⛔	⛔
-		↘	↘		M		⛔	⛔	⛔	m		⛔	⛔	⛔
.		↘	↘		N	▶	▶	▶	⛔	n	▶	▶	▶	▶
/		↘	↘		O		⛔	⛔	⛔	o		⛔	⛔	⛔
0	▶	▶	▶	▶	P		⛔	⛔	⛔	p		⛔	⛔	⛔
1	▶	▶◁	▶	▶◁	Q		⛔	⛔	⛔	q		⛔	⛔	⛔
2	▶	▶◁	▶	▶◁	R		⛔	⛔	⛔	r	▶	▶	▶	▶
3	▶	▶◁	▶	▶◁	S		▶	▶	⛔	s		▶	▶	⛔
4	▶	▶◁	▶	▶◁	T		⛔	⛔	⛔	t	▶	▶	▶	▶
5	▶	▶◁	▶	▶◁	U	▶	▶	▶	⛔	u	▶	▶	▶	⛔
6	▶	▶◁	▶	▶◁	V		⛔	⛔	⛔	v	▶	▶	▶	▶
7	▶	▶◁	▶	▶◁	W		▶	▶	⛔	w		▶	▶	⛔
8		◁	⛔	◁	X		⛔	⛔	⛔	x	▶	▶	▶	⛔
9		◁	⛔	◁	Y		⛔	⛔	⛔	y		⛔	⛔	⛔
:		↘	↘		Z		⚓	⛔	⛔	z		⛔	⛔	⛔
;		↘	↘		[		↘	↘		{		↘	↘	
<		↘	↘		\	↘	↘	↘	↘	|		↘	↘	
=		↘	↘		]		↘	↘		}		↘	↘	
>		↘	↘		^		↘	↘		~		↘	↘	
?		↘	↘		_		↘	↘		DEL		↘	↘	

Legenda:

- ↘ Bescherming van het erop volgende teken: `\@` wordt als het `@`-teken opgevat
- ▶ Transformatie: `\b` representeert de backspace (ASCII-teken 8)
- ⚓ Anker: `\b` geeft een woordgrens aan
- ◁ Referentie: `\1` verwijst naar groep `1`
- ⛔ Verbod: `\B` is niet toegestaan

Waar niets in een kolom is ingevuld, wordt de tekencombinatie letterlijk genomen: `\A` in een string literal wordt dus geïnterpreteerd als een backslash gevolgd door de hoofdletter A.

Kleine complicatie die ik bij mijn experimenten ben tegengekomen: in een raw string wordt de backslash altijd letterlijk als zodanig opgevat, behalve als deze het laatste teken tussen de aanhalingstekens is. Met `r"\A\"` krijg ik de foutmelding “SyntaxError: incomplete input”. Een echt elegante oplossing weet ik hier niet voor, want `r"\A\\"` wordt opgevat als backslash–A–backslash–backslash oftewel `\A\\`. Niet fraai maar wel afdoende is `r"\A""\\"` of `r"\A" "\\"`, dat uiteindelijk hetzelfde betekent als `r"\A" + "\\"` maar tenminste in compileertijd wordt samengesteld tot `\A\`.

Een ander ongemakje is de ogenschijnlijke dubbelzinnigheid van de combinaties van een backslash met een cijfer. Waar in een kolom de combinatie ▶◁ voorkomt, gelden de volgende regels binnen een raw string:

`\0`	Octale aanduiding van ASCII-teken 0.
`\1` tot en met `\99`	Decimale verwijzing naar groep `1` t/m `99`.
`\00` tot en met `\07`	Octale aanduiding van ASCII-teken 0 t/m 7.
`\000` tot en met `\377`	Octale aanduiding van (extended) ASCII-teken 0 t/m 255.
`\400` tot en met `\777`	Ongeldige octale waarde voor een teken uit de extended ASCII-tabel.

Botsingen zijn eenvoudig te vermijden. Met het patroon `\100` vind je een `@`-teken indien dit in de te onderzoeken string voorkomt. Zoek daarom naar `\10[0]` of `\10\x30` of `\10\060` als je in plaats daarvan de inhoud van groep `10` gevolgd door het cijfer `0` wilt opsporen. In een als raw string aangeboden vervanging kun je `\g<10>0` of `\10\060` schrijven — `\10\x30` is verrassend genoeg verboden!

Misschien is het niet zo’n geweldig idee om een vervanging in de vorm van een raw string op te stellen. Binnen een gewone string literal (dus `"..."` i.p.v. `r"..."`) zouden in bovenstaand geval onder meer de volgende uitdrukkingen kunnen worden gebezigd: `\g<10>0`, `\g<10>\060`, `\\g<10>\\060`, `\g<10>\x30`, `\g<10>\u0030`, `\g<10>\U00000030` en `\g<10>\N{digit zero}`. Behalve de eerste van dit stel zijn alle varianten nogal onzinnig, maar ze demonstreren wel dat je ‘moeilijke’ tekens zoals het NULL-teken en het Chinese 人-karakter in de vervanging kunt onderbrengen. Verwijs liever niet naar een groep met uitdrukkingen zoals `\\10`. Op zichzelf staand zou dit inderdaad werken, maar in de combinatie `\\10\x30` krijg je toch weer dat `@`-teken als gesubstitueerde waarde terug.

Realiseer je bij dit alles dat het *repl*-argument van de `sub`- of `subn`-functie of -methode twee keer wordt geëvalueerd:

- De eerste keer vóórdat `sub` of `subn` ermee aan de slag gaat. Is het een variabele of de aanroep van een callable, dan wordt de waarde ontvangen die uit de evaluatie daarvan resulteert (bij een aanroep dus de returnwaarde). Is het een gewone string literal, dan vinden er substituties plaats volgens kolom **Quot**. Is het een raw string, dan wordt alles tussen de aanhalingstekens letterlijk genomen, behalve dat ene geval dat ik hierboven heb beschreven. Is het een referentie naar een callable, dan wordt het betreffende callable object ontvangen.
- Ten tweeden male wanneer `sub` of `subn` de ontvangen *repl*-waarde gaat toepassen. Is dat een stringwaarde, dan wordt deze geëvalueerd volgens kolom **Repl**. Is het een callable object, dan wordt deze aangeroepen zoals in paragraaf 3.1 beschreven. De resulterende stringwaarde wordt letterlijk genomen (alle gewenste omzettingen moeten binnen de callable zijn gedaan).

Genoeg nu. Ik zet er een streep onder:

```
>>> print("\137\N{low line}\U0000005f\u005f\x5f")
_____
```

Appendix D Referenties

[1] Pieter Spronck, *De programmeursleerling — Leren coderen met Python 3*, versie 1.1.0, 4 juni 2023. http://www.spronck.net/pythonbook/dutchindex.xhtml.

[2] Python docs, "re — Regular expression operations" in *The Python Standard Library*. https://docs.python.org/3/library/re.html.

[3] Python docs, "Regular Expression HOWTO" in *Python HOWTOs*. https://docs.python.org/3/howto/regex.html#regex-howto.

[4] Meindert Meindertsma, "Reguliere expressies" in *Een jaar of wat*. https://home.hccnet.nl/demsmsma/ejow/ejow.html#%5B%5BReguliere%20expressies%5D%5D.

[5] Wikipedia, *Regular expression*. https://en.wikipedia.org/wiki/Regular_expression

[6] Meindert Meindertsma, *Reguliere expressies behapbaar maken — "Verdeel en heers"*, 1 en 8 juni 2023, bijgewerkt in februari 2024. [Spronck-H25.pdf].