Mestrado em Engenharia Informática
Engenharia Gramatical Março, 2024

TP1 - Beautifiers

Grupo 4

Vasco Oliveira (pg54269), João Loureiro (pg53924) and Luís Fernandes (pg54019)

Resumo. Esta monografia aborda a utilização de *beautifiers*, ferramentas destinadas a melhorar a estética e organização de códigos fonte, interfaces gráficas e documentos. O foco principal é em *beautifiers* relacionados à formatação de código, com ênfase em formatadores de código Python, como Black e Autopep8, e as ferramentas CLanfFormat e Prettier, utilizadas para diversas linguagens.

Palavras-chave: Beautifiers, Formatação de Código, PEP 8, Autopep8, Black, Prettier, CLanfFormat.

Table of Contents

1 Definição de *Beautifiers*

Beautifiers, ou "embelezadores" em português, são ferramentas, técnicas ou processos utilizados para melhorar a estética, legibilidade e organização de códigos fonte, interfaces gráficas, documentos ou qualquer elemento visual relacionado ao desenvolvimento de software.

Esta monografia irá se focar principalmente na categoria de *beautifiers* relacionados à formatação do código. Nesse contexto, as ferramentas têm como objetivo automatizar o ajuste da indentação do código fonte sem alterar a funcionalidade do código, tornando-o mais legível e consistente.

2 Ferramentas

2.1 Formatadores de Código Python

Black e *Autopep8* são ambas ótimas ferramentas para formatar código Python. *Black* é a ferramenta mais popular, baseado pela sua atividade no *GitHub*, enquanto que a ferramenta *Autopep8* fica um bocado atrás na popularidade.

Ambos *Black* e *Autopep8* seguem as diretrizes e convenções estabelecidas pelo guia de estilo de código Python *PEP 8*, mas têm abordagens ligeiramente diferentes.

O *Black* é conhecido por ser inflexivo, no sentido que adota uma abordagem mais rigorosa e consistente em relação à formatação do código. Ele não oferece muitas opções de configuração e procura impor uma única maneira de formatar o código. *Autopep8* é mais configurável, permitindo personalizar algumas opções de formatação de acordo com as preferências e objetivos de cada pessoa.

2.1.1 PEP 8 - Guia de Estilo para Código Python

PEP 8 (Python Enhancement Proposal 8), é o guia de estilo oficial para código Python. Foi escrito por Guido van Rossum, o criador da linguagem Python, e descreve as convenções de formatação que os programadores de Python devem seguir para escrever um código claro, legível e consistente. Aqui estão alguns pontos-chave do PEP 8:

1. Indentação: Usa 4 espaços por cada nível de indentação.

   Exemplo:

```
1     # Add 4 spaces (an extra level of indentation) to
    distinguish arguments from the rest.
2     def long_function_name(
3         var_one, var_two, var_three,
4         var_four):
5     print(var_one)
6
```

Listing 1: Identação

2. Comprimento das Linhas: Linhas devem ter no máximo 79 caracteres (ou 72 para docstrings). Se uma expressão não couber em uma única linha, use parênteses para dividir.

   Exemplo:

```
    resultado = (2000000002 +
                 2000000002 +
                 2000000002 +
                 2000000002)

```

Listing 2: Comprimento de Linhas

3. Os *imports* devem ser agrupadas na seguinte ordem:

   (a) *Imports* da biblioteca padrão.

   (b) *Imports* relacionados com terceiros.

   (c) *Imports* de especificas aplicações/bibliotecas locais.

   Deve-se adicionar uma linha em branco entre cada grupo de importações.

4. Evite espaços em branco desnecessários, como nas seguintes situações:

   (a) Imediatamente dentro de todo o tipo de parênteses.
   `spam( ham[ 1 ], { eggs:  2 } )` → `spam(ham[1], {eggs:  2})`

   (b) Entre uma vírgula final e um parêntese de fechamento subsequente.
   `foo = (0, )` → `foo = (0,)`

   (c) Imediatamente antes de uma vírgula, ponto e vírgula ou dois pontos.
   `print(x , y) ; x` → `print(x, y); x`

É possível ver estas regras apresentadas e o resto delas no website oficial do *PEP 8* [1].

2.1.2 Exemplos de Uso

De seguida apresenta-se um exemplo de como fica o código depois de ser formatado pelo *Black* e o *AutoPep8*, sem nenhumas alterações nas opções das ferramentas:

```
def weird_function(x, y):
    z = x +             y; return z

class WeirdClass             :
    def __init__(self, name, number):


        self.name = name; self.number = number
        self.color = "yellow"

def greet(self): print(f'Hello, {self.name}!')
```

```

if __name__ == "__main__":       w_class        = WeirdClass("Weirdo"
    )
w_class.greet()
```

Listing 3: Original

```
def weird_function(x, y):
    z = x + y
    return z


class WeirdClass:
    def __init__(self, name, number):

        self.name = name
        self.number = number
        self.color = "yellow"


def greet(self):
    print(f"Hello, {self.name}!")


if __name__ == "__main__":
    w_class = WeirdClass("Weirdo")
w_class.greet()
```

Listing 4: Black

```
def weird_function(x, y):
    z = x + y
    return z


class WeirdClass:
    def __init__(self, name, number):

        self.name = name
        self.number = number
        self.color = "yellow"


def greet(self): print(f'Hello, {self.name}!')


if __name__ == "__main__":
    w_class = WeirdClass("Weirdo")
w_class.greet()
```

Listing 5: AutoPep8

2.2 Vantagens e Desvantagens

Apesar do *Black* e do *Autopep8* serem ferramentas muito similares, cada uma tem as suas vantagens e desvantagens distintias, podendo ser mais vantajoso usá-las em diferentes contextos dentro da formatação de código Python.

2.2.1 Black

Vantagens:

- **Formato Consistente:** Como o *Black* aplica um formato consistente automaticamente, ajuda a manter a consistência para, por exemplo, projetos grandes em que cada pessoa tem o seu estilo de programação.
- **Inicialização Fácil**: É uma ferramenta de uso extremamente simples, tendo simplesmente de abrir um terminal e usar o comando "black (nome do ficheiro python)" para ser aplicada.
- **Opções Extra**: Apesar da ferramenta *Black* ser bastante restrita na maneira como formata o código, ainda oferece uma variedade de opções para melhorar a experiência do utilizador, como por exemplo a opção "–fast", que possivelmente compromete a segurança do código para ter uma performance mais rápido, ou a opção "–line-ranges", onde só formata as linhas que foram especificadas, ou a opção "–line-length", que especifica o limite de caracteres por linha (sendo 88 o valor padrão).

Desvantagens:

- **Personalização Limitada**: A falta de opções de configuração, em comparação com outras ferramentas como o *Autopep8*, pode ser uma desvantagem para desenvolvedores que preferem um maior controlo sobre o estilo de formatação.
- **Formatação Radical**: o *Black* é bastante radical ao formatar o código original, não havendo uma opção direta para salvar o código formatado num ficheiro diferente do original.

2.2.2 Autopep8

Vantagens:

- **Configurável**: o *Autopep8* é mais configurável do que o Black, permitindo que os desenvolvedores personalizem mais opções de formatação de acordo com suas preferências, principalmente através da opção "–ignore", em que é usada para desativar certas regras durante a formatação.
- **Formatação Menos Radical**: o *Autopep8*, ao contrário do *Black*, tem a opção de salvar o código formatado num ficheiro diferente do original, não obrigando o desenvolvedor a alterar o código original.

Desvantagens:

- **Menos Rigoroso**: Por ser mais configurável, pode haver menos rigor na aplicação consistente de um estilo de formatação, o que pode levar a disparidades no código de diferentes desenvolvedores.
- **Inicialização Mais Complexa**: Embora o uso do *Autopep8* também seja intuitivo, tem mais um passo que o *Black* para se poder formatar realmente o código tendo que especificar qual ficheiro terá o output da formatação ou usar a opção –in-place para formatar o ficheiro orginal, ficando assim o comando: "autopep8 (nome do ficheiro python) (–in-place/nome do ficheiro output)", a omissão da opção –in-place ou do ficheiro output faz com que o *Autopep8* devolva o resultado imprimido no terminal.

3 Prettier

O *Prettier* é um formatador de código "opinado", isto é, que ignora o estilo de escrita original e impõe um estilo de código consistente em todo o código escrito pelo o utilizador, reescrevendo-o com as suas próprias regras que consideram o tamanho máximo de cada linha (*code wrapping*). [2]

Suporta várias linguagens, dentro das quais: *JavaScript, JSX, Angular, Vue, Flow, TypeScript, CSS, Less, SCSS, HTML, Ember/Handlebars, JSON, GraphQL, Markdown* (incluindo *GFM* e *MDX v1*), e *YAML.* Fornece integração com a maioria dos editores de texto e *IDEs* (como o *Visual Studio* e o *VS Code*) e ajuda a poupar tempo e energia ao formatar automaticamente código quando o código é guardado, eliminando a necessidade de rever o código manualmente.

Permite também a configuração de alguns parâmetros como o tamanho de espaços, *tabs*, posição de chavetas e *brackets* (isto é, se ficam na mesma linha ou se passam para a linha seguinte), *end-of-lines*, quantidade de atributos por linha, entre outros...

Ao contrário dos *beautifiers* tradicionais, o *Prettier* não apresenta muitas mais opções de configuração e de personalização, nem será atualizado no futuro para adicionar novas opções. Isto porque a equipa responsável pelo a sua criação desenvolveu o *Prettier* com um objetivo: **uniformizar a estilo de escrita de código e terminar com os debates sobre qual é o melhor estilo**, daí ser um formatador "opinado". Dito isto, o *Prettier* é atualizado com frequência, não estando de todo abandonado.

3.1 Aspetos Técnicos

O *Prettier* pega na *AST* (*Abstract Syntax Tree*), a representação em árvore do código *source* de um programa que expressa a estrutura deste, e usa o para gerar uma *string*, o que permite ao *printer* "medir" o tamanho e ver se esta vai caber numa linha ou se é preciso quebrá-la.

Assim, grande parte da lógica na impressão de uma *AST* envolve em gerar uma representação abstrata do *output* formatado, incluindo alguns comandos específicos. Por exemplo: `["(", line, arg, line, ")"]` representa uma concatenação de abrir parênteses, um argumento e fecho dos parênteses, mas se isto tudo não couber numa só linha o *printer* pode quebrar onde `line` está especificado.

3.2 Vantagens e Desvantagens

Como qualquer tipo de ferramenta, o *Prettier* oferece vantagens e desvantagens aos seus utilizadores. Começando pelas vantagens:

- **Estilo Pré-Definido:** como mencionado anteriormente, o *Prettier* impõe um estilo de formatação ao código escrito, o que é ótimo para escritores de código principiantes e para normalizar e uniformizar todo o código escrito numa dada equipa.
- **Fácil de adotar:** o estilo imposto pelo o *Prettier* foi escolhido e aprimorado com a ajuda de vários utilizadores, pelo o que é considerado por grande parte dos seus utilizadores como fácil de adotar.
- **Organizar código já existente:** o *Prettier* é fácil de aplicar a código já existente de forma a embelezar e corrigir algumas falhas de estéticas que possam existir nestes.
- ***Plugins* para outras línguas:** para além das línguas previamente mencionadas, existem também *plugins* oficiais e desenvolvidos pela a comunidade que permitem utilizar o *Prettier* em outras linguagens, dentro das quais: *Java, Kotlin, SQL, jinja, Rust, TOML,* etc...

Já em termos de desvantagens:

- **Pouca flexibilidade:** como já referido anteriormente, é um formatador de código "opinado", não permitindo ao utilizador personalizar a forma como o código é formatado, o que pode ser uma desvantagem para aqueles que preferem uma experiência mais personalizada ou mesmo para quem não gosta da forma de como o *Prettier* formata o código.
- **Falta de suporte para algumas línguas populares:** na documentação oficial do *Prettier*[2] não encontramos referências a *plugins* oficialmente apoiados para linguagens populares como *C, C++* e *Python*, pelo o que programadores destas linguagens devem utilizar outro formatador.

3.3 Exemplos de uso

Seguem-se então alguns exemplos do resultado de alguns excertos de código desformatados e o seu resultado após o *Prettier* o formatar.

```
function HelloWorld({greeting = "hello", greeted = '"World"',
    silent = false, onMouseOver,}) {

  if(!greeting){return null};

     // TODO: Don't use random in render
  let num = Math.floor (Math.random() * 1E+7).toString().replace
    (/\.\d+/ig, "")

  return <div className='HelloWorld' title={`You are visitor number
    ${ num }`} onMouseOver={onMouseOver}>
```

```
9
10      <strong>{ greeting.slice( 0, 1 ).toUpperCase() + greeting.slice
        (1).toLowerCase() }</strong>
11      {greeting.endsWith(",") ? " " : <span style={{color: '\grey'}}>
        ", "</span> }
12      <em>
13    { greeted }
14    </em>
15      { (silent)
16        ? "."
17        : "!"}
18
19      </div>;return "Formatado!"
20  }
```

Listing 6: Original (JavaScript)

```
1  function HelloWorld({
2    greeting = "hello",
3    greeted = '"World"',
4    silent = false,
5    onMouseOver,
6  }) {
7    if (!greeting) {
8      return null;
9    }
10
11   // TODO: Don't use random in render
12   let num = Math.floor(Math.random() * 1e7)
13     .toString()
14     .replace(/\.\d+/gi, "");
15
16   return (
17     <div
18       className="HelloWorld"
19       title={`You are visitor number ${num}`}
20       onMouseOver={onMouseOver}
21     >
22       <strong>
23         {greeting.slice(0, 1).toUpperCase() + greeting.slice(1).
       toLowerCase()}
24       </strong>
25       {greeting.endsWith(",") ? (
26         " "
27       ) : (
28         <span style={{ color: "grey" }}>", "</span>
29       )}
30       <em>{greeted}</em>
31       {silent ? "." : "!"}
32     </div>
33   );
34   return "Formatado!";
```

```
}
```

Listing 7: Formatado (JavaScript)

```
---
invoice    :    34843
date    :     2001-01-23
bill-to:     &id001
    given     : Chris
    family   : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city          : Royal Oak
        state        : MI
        postal    : 48046
ship-to: *id001
product:
    -

      sku          : BL394D
      ? quantity
      : 4
      description : Basketball
      ? price
      : 450.00


    -
      sku           :    BL4438H
      quantity       :   1
      description:        Super Hoop
      price          :   2392.00


tax   :   251.42
total : 4443.52
comments: >
    Late afternoon is best.
    Backup contact is Nancy
    Billsmer @ 338-4338.
```

Listing 8: Original (YAML)

```
---
invoice: 34843
date: 2001-01-23
```

```yaml
bill-to: &id001
  given: Chris
  family: Dumars
  address:
    lines: |
      458 Walkman Dr.
      Suite #292
    city: Royal Oak
    state: MI
    postal: 48046
ship-to: *id001
product:
  - sku: BL394D
    quantity: 4
    description: Basketball
    price: 450.00

  - sku: BL4438H
    quantity: 1
    description: Super Hoop
    price: 2392.00

tax: 251.42
total: 4443.52
comments: >
  Late afternoon is best.
  Backup contact is Nancy
  Billsmer @ 338-4338.
```

Listing 9: Formatado (YAML)

4 CLangFormat

O *clangFormat* é uma ferramenta de formatação de código fonte para os idiomas *C*, *C++*, *Objective-C* e *Objective-C++*. Ele faz parte do projeto *LLVM* (para saber mais verifique a pagina oficial do LLVM[3]) e é uma ferramenta muito útil para garantir a consistência e a legibilidade do código fonte.

De seguida, estão representadas algumas *features* que o *clangFormat* oferece aos utilizadores:

- **Conformidade com Padrões de Estilo:** Pode ser configurado para formatar o código de acordo com diferentes padrões de estilo, como o estilo do LLVM, Google, GNU e muitos outros.
- **Configuração Flexível:** Oferece uma ampla gama de opções de configuração que podem ser personalizadas para atender às preferências de formatação específicas de

um projeto. Isso inclui opções para espaçamento, indentação, quebras de linha, alinhamento e muito mais.

- **Integração com Editores e IDEs:** Pode ser facilmente integrado em vários editores de texto e IDEs, como Visual Studio Code, Sublime Text, Emacs e Vim. *CLion* permite automaticamente o *ClangFormat*, e aplica as configurações de um ficheiro *.clang-format* caso exista [4].
- **Utilização em Linha de Comando:** Além da integração com editores e *IDEs*, o *clangFormat* pode ser usado diretamente a partir da linha de comando.
- **Preservação de Comentários e Formatação Seletiva:** Preserva comentários existentes no código fonte e oferece opções para formatar seletivamente partes específicas do código.
- **Reconhecimento de Estilo Automático:** Inferir automaticamente o estilo de formatação a partir de um código fonte existente, tornando mais fácil a configuração inicial da ferramenta em projetos já existentes.

4.1 Ficheiro de configuração

Ficheiro *.clang-format* com convenções de formatação devem ser colocados na árvore do projeto. Cada um deles é aplicado nos ficheiro de código no mesmo nível e nas subdiretorias. Ter múltiplos ficheiros *.clang-format* para um único projeto, permite usar diferentes estilos para pastas de código diferentes.

ClangFormat oferece muitas opções de estilo, que podem ser misturadas para criar o conjunto de regras mais adequado para uma equipa. Também fornece vários estilos integrados para *C/C++*: *LLVM*, *Google*, *WebKit*, *Chromium* e *Mozilla*. Esses estilos predefinidos podem ser usados como estão ou como base para o seu conjunto personalizado de definições de estilo.

```
1
2  # The configuration options will be based on the chosen
       style.
3  BasedOnStyle: LLVM
4
5  # IndentWidth: Number of spaces for each indentation
       level.
6  IndentWidth: 4
7
8  # UseTab: Use tabs for indentation instead of spaces.
9  UseTab: Never
10
11 # TabWidth: Number of spaces equivalent to one tab when
       'UseTab' is set to 'Never'.
12 TabWidth: 4
13
```

```
# AccessModifierOffset: Indent level for access modifiers.
AccessModifierOffset: -2

# ConstructorInitializerIndentWidth: Additional indentation for constructor initializers.
ConstructorInitializerIndentWidth: 4

# AllowShortFunctionsOnASingleLine: Allow short functions to be written on a single line.
AllowShortFunctionsOnASingleLine: All

# SpaceBeforeAssignmentOperators: Add a space before assignment operators
SpaceBeforeAssignmentOperators: true

# PointerAlignment: Alignment style for pointer and reference operators (* and &).
PointerAlignment: Right

# BreakBeforeBraces: Whether to break before opening braces for control statements and functions.
BreakBeforeBraces: Allman

# CommentPragmas: Keep or remove pragma directives in comments.
CommentPragmas: Keep
```

Listing 10: Ficheiro de configuração .clang-format

4.2 Vantagens e Desvantagens

Tal como visto nas ferramentas anteriores, o *ClangFormat* também apresenta as suas vantagens e desvantagens na sua utilização. Comecemos pelas desvantagens:

- **Curva de Aprendizagem:** Configurar e compreender as várias opções requer alguma aprendizagem inicial, especialmente em equipas com pouco conhecimento sobre ferramentas de formatação;
- **Limitações de Configuração:** Mesmo com o seu grande leque de opções de configuração, alguns formatos em especifico podem ser bastante difíceis de configurar, dependendo da sua complexidade;
- **Desafios de *Override*:** Em alguns casos, os programadores podem necessitar de *override*, a formatação automática do *ClangFormat*, o que pode adicionar complexidade ao trabalho.

Após abordar as desvantagens da sua utilização, falta verificar quais vantagens é que a ferramenta apresenta ao utilizador:

- **Consistência:** Assegura consistência na formatação do código ao longo do projeto, facilitando assim o trabalho de leitura e compressão dos programadores.
- **Poupar tempo:** Com a formatação automática, os programadores não perderam tempo em formatar o código manualmente.
- **Integridade:** Pode ser integrado em diferentes meios de desenvolvimento, IDEs entre outros sistemas. Assegura que a formatação do código é mantida ao longo do fluxo de trabalho.

5 Conclusão

Com a resolução deste trabalho conseguimos expandir o nosso conhecimento sobre ferramentas de análise de código, neste caso *Beautifiers*, bem como um entendimento mais aprofundado destas ferramentas que muitas pessoas na área de informática utilizam diariamente.

Através das ferramentas analisadas, conseguimos perceber que cada uma possui as suas vantagens e desvantagens distintas, e portanto, utilidades diferentes. Umas ferramentas permitem uma experiência de utilização mais consistente e fácil de utilizar mas com um menor grau de personalização (como o *Prettier*), enquanto que outras permitem uma experiência mais personalizada ao custo de ser difícil de configurar, sendo mais adequada para programadores mais experientes (como o *CLangFormat*), para além de cada uma se adequar a linguagens de programação diferentes.

References

[1] Guia oficial do *PEP 8* `https://peps.python.org/pep-0008/`

[2] Documentação oficial do *Prettier* `https://prettier.io/docs/en/`

[3] Página oficial do *LLVM* `https://llvm.org/`

[4] *CLangFormat* no *CLion* `https://www.jetbrains.com/help/clion/clangformat-as-alternative-formatter.html`