Ryan Burgert 110176886
CSE526 Final Project Writeup May 11 2021

* Small note: all of this text in this document is also contained inside of final_writeup.txt, in case you want to try copy/pasting any of the completions in this document, or if you just prefer reading things in text editors (I know I do).

Part 1: Problem and Plan

Writing code is a repetitive process. Most languages require you to write the same things over and over again. Some text editors try to relieve this burden by adding autocompletion, which reduces the time it takes to write variable names and keywords, often involving choosing from a menu of options.

Some other editors, like sublime, go a little bit further. Sublime, for example, will automatically insert a) each time you insert a (. It also automatically preserves the indent each time you press the enter key so you can avoid pressing tab multiple times each time you start a new line. This type of completion, which doesn't involve choosing from a menu of options, doesn't have an official name. I choose to call them 'microcompletions', as they help you write code faster but are distinct from autocompletions.

My project aims to take these microcompletions more seriously, giving the user a set of rules that will allow them to write code much more efficiently and with less effort.

This is where my IDE comes in. I've created an IDE for python, called 'rp'. It's completely terminal based, and it features hundreds of different microcompletions. For example, some help create python functions, help to write array literals, or help you avoid needing to press the shift or arrow keys. The goal is to minimize the effort needed to write code in python.

Effort can be thought of as the number of keystrokes it takes to write a given piece of code, as well as how easy those keys are to reach on the keyboard. For example, entering the character) is harder than entering the character 0, because) involves two keystrokes: the shift key and the 0 key. Similarly, pressing the spacebar is preferable to pressing the comma key, because the spacebar is a larger target and thus easier to hit with fingers from either one of your hands.

However, the way rp's microcompletion engine is currently written, it makes it impossible to perform unit tests. When adding a new microcompletion rule, the only way to know if I'm breaking anything is to manually try out all of the use cases. Additionaly, rp's current engine cannot be used for any program other than rp - meaning that without a new engine, there's no way that these microcompletions could be brought to any other editor such as sublime or vim. (Although this project doesn't implement completions for any of these editors, the functional framework which it proposes opens the door to reusing this code to talk to these editors via a socket connection - similar to how the YouCompleteMe or JEDI autocompletion engines work).

This is where this project comes in. The goal of my final project for CSE526 is to create a functional microcompletion engine, that can be unit tested, is self-documenting, and is modular enough to be eventually ported to other editors.

Part 2: Design

Part 2A: Formal definition of the microcompletion engine

A microcompletion, in it's most basic form, is what happens to text in an editor when you press a key on your keyboard. Microcompletions can be as simple as inserting a space character when you press the space bar, or as advanced as creating a new function when pressing the space bar. These are both microcompletions.

Formally, a microcompletion is a function that takes in two parameters (an editor state and a keystroke) and returns an editor state. An editor state is what you see when you open a text editor, consisting of two values: text (a string) and the cursor position (an integer). The microcompletion engine I implemented for this class doesn't currently take into account more advanced aspects of the editor state such as text selections or multiple cursors, and instead assumes we have a single text cursor - but these can be added in a future version.

Part 2B: Microcompletion notation

To specify how microcompletions should behave, I've created a formal way of expressing examples of their usages. These expressions are put into the docstrings of every microcompletion rule (each of which is expressed as a python function). These expressions are both used for documentation, and for deriving unit tests.

It is easier to explain how this works by showing examples, so thats that this section will do.

Each expression consists of a series of editor states, separated by keystrokes.
Recall that an editor state consists of a string and a cusor position.
Here is an editor state of the string "Hello World", with the cursor right after Hello:

```
‹"Hello¦ World!"›
```

Notice how the ¦ represents the cursor position, and how each editor state is surrounded by ‹ and › characters. These are unicode characters which are easily typed on a macbook, but are unlikely to appear in python code relevant to any microcompletion rules.

Now, for an example of a microcompletion expression. In the engine code that I've submitted, entering the (character automatically inserts the) character, like many other text editors such as sublime. Here's how that rule will be written:

```
Matching parenthesis: adding ( also adds )
|   ‹¦›   (   ‹(¦)›
```

This means that if we start in the ‹¦› editor state (aka an empty document), then we press the (keystroke, then it gets turned into the ‹(¦)› editor state (aka two parenthesis surrounding the cursor).

Here is another, more mundane example of using the arrow keys:

```
Using the left arrow key and backspace
|   ‹"Hello¦">   left  left   ‹"Hel¦lo">   backspace  ‹"He¦lo">
```

Note how the top line represents the title of the completion, and each line of the rule starts with |.

Note that we can add additional tests to a given expression by separating them with '...':

```
Matching parenthesis: adding ( also adds )
|   ‹¦›   (   ‹(¦)›
...
|  ‹(¦)›  (   ‹((¦))›


Backspacing Parenthesis: deltes both () at the same time
|  ‹((¦))›   backspace   ‹(¦)›    backspace    ‹¦›
```

Also note that these expressions can be done for multiple lines of code at once:

```
Pressing the enter key preserves the indent:
|  ‹if True:›                              ‹if True:›
|  ‹    print("Hello")¦›      enter       ‹    print("Hello")›
|                                         ‹    ¦›
```

Some completions are used to illustrate edge cases that shouldn't happen. When an expression is SUPPOSED to fail, the description starts with 'SHOULD_FAIL'

```
SHOULD_FAIL! Pressing 'x' doesn't insert 'y'. Duh.
|  ‹"¦">     x    ‹"y¦">


Instead, it should insert x
|  ‹"¦">     x    ‹"x¦">
```

Part 2C: Microcompletion motifs and rule examples

For my project, I've implemented over 50 different rules, totalling in over 1500 lines of python code (most of which is documetation). This sounds overwhelming at first, but it's actually not. Many of the rules share similar recurring themes, which I call "motifs". By learning just a few motifs, hundreds of microcompletion rules (many of which are designed to work together) suddenly feel very natural and intuitive.

One of the largest recurring motifs is the following: when what the user would have typed normally would result in invalid python syntax, it's an oppurtunity to do something interesting. In fact, most other motifs stem from this one: syntax-breaking keystrokes can be used as commands, because they would otherwise be useless.

For example, there is no situation in python where '++' is a valid token, which means it's an oppurunity to add something new - '++' turns into '+=1', saving a keystroke

```
Adds the ++ and -- operators to python
|   ‹x¦›   + +   ‹x+=1¦›
...
|   ‹x¦›   - -   ‹x-=1¦›
...
|   ‹x¦›   +   ‹x+¦›   +    ‹x+=1¦›
```

For another example, another "motif" is that when it's possible, avoid using the shift key. One consequence of this motif are the following completion rules:

```
Eight-to-star: When possible, treat 8 like the * charcter to avoid
pressing shift
|   ‹def f(¦):›     8   ‹def f(*¦):›
...
|   ‹def f(x,¦):›   8   ‹def f(x,*¦):›   8   ‹def f(x,**¦):›


Eight-to-star: Works for calling functions too, because '8x' is an
invalid python token
|   ‹print(¦)›       8 x     ‹print(*x¦)›
...
|   ‹print(¦)›       8       ‹print(8¦)›      x      ‹print(*x¦)›


Semicolon-to-colon: array[;;-1] is not valid syntax, but array[::-1] is
|   ‹array[¦]›       ;     ‹array[:¦]›
...
|   ‹array[¦]›    ;  ;     ‹array[::¦]›


Three-to-comment: '3comment' is invalid syntax, but '#comment' is
|   ‹3›       c     ‹#c¦›


Two-to-decorator: '2property' is invalid syntax, but '@property' is
|   ‹2›       p     ‹@p¦›


Two-to-decorator: you can't decorate integers, but you can decorate
decorators
|   ‹@decorator›         ‹@decorator›
|   ‹¦›             2    ‹@¦›


Dash-to-underscore: identifiers can't start with -, but can with _
```

```
|   ‹def f(¦):›     –   ‹def f(_¦):›
...
|   ‹for ¦ in x:›   –   ‹for _¦ in x:›
...
|   ‹import ¦›    –   ‹import _¦›
```

Another motif is that .= [= and)= are all valid operators. x*=y means x=x*y, and so x.=y means x=x.y. Similarly, x)=y means x=y(x) and x[=i means x=x[i]

```
Dot-equals operator: The .= operator activates upon pressing =
|   ‹matrix.¦›   =   ‹matrix=matrix.¦›


Call-equals operator: The )= operator
|   ‹value)¦›   =   ‹value=¦(value)›   f u n c   ‹value=func¦(value)›


Index-equals operator: The [= operator
|   ‹array¦›     [  =    ‹array=array[¦]›
```

Another motif is that the spacebar does a lot of work. Upon pressing space, keywords are completed, and functions can be declared, etc. Please note that '␣' is equivalent to 'space', which means the spacebar.

```
D-to-def: You can write out entire function declarations without ever
pressing : , ( or )
|   ‹d¦›   space   ‹def ¦():›
...
|   ‹d¦›   ␣       ‹def ¦():›
...
|   ‹def func¦():›   ␣   ‹def func(¦):›
...
|   ‹def func(x¦):›  ␣  ‹def func(x,¦):›
...
|   ‹¦›    d ␣ f ␣ x ␣ y    ‹def f(x,y¦):›


F-to-for: For loops can be written quickly using the spacebar
|   ‹f¦›   ␣   ‹for ¦ in :›
...
|   ‹for x¦ in :›   ␣   ‹for x in ¦:›
...
|   ‹¦›     f ␣ x ␣ y     ‹for x in y¦:›


R-to-return:
|   ‹def f():›         ‹def f():›
|   ‹    r¦›      ␣    ‹    return ¦›
```

It is true that it doesn't always make sense to turn r to return, or d to def etc. For example, 'r and d' is valid syntax as well. However, there are multiple ways to get around this problem, and the

majority of the time turning r into return is more useful. RP's current completion engine takes this all into account, but the one I made for this final project doesn't do that yet. One possible solution could be:

```
R-and-not-return:
|   ‹def f():›         ‹def f():›                  ‹def f():›
‹def f():›
|   ‹    r¦›      ␣    ‹    return ¦›     a n d    ‹    return and¦›
␣    ‹    r and ¦›
...
|   ‹def f():›                     ‹def f():›
|   ‹    r¦›      r ␣ a n d ␣ d    ‹    r and d¦›
```

There are many other motifs as well, and many of them are really cool and fun to talk about and really useful to know. But to keep this writeup from taking too many pages, I'll move on to the next part.

Part 3 and part 4: Implementation and testing

Note that I'm treating part 3 and part 4 as the same: A huge point of this program is to create a testing suite for the microcompletion rules, so testing basically IS the implemetation.

In my github repository, there is a 'README.txt' that tells you how to run my program. This details most of the implementation details, so in this section I'll talk a little about how my code is structured. This details the libraries that were used and how to install them etc.

The microcompletion rules are very modular. Each rule is contained inside a separate function, which has a docstring telling you how it works.

Here are a few simplified examples relating to rules we've talked about so far:

```
@engine.add_rule
def preserve_indent(state,keystroke):
      """
      Hitting enter can preserve the indent
      |   ‹def f():›              ‹def f():›
      |   ‹    print()¦›   enter   ‹    print()›
      |                           ‹    ¦›

      SHOULD_FAIL! Hitting enter doesn't simply go to the beginning of
the line with this rule
      |   ‹def f():›              ‹def f():›
      |   ‹    print()¦›   enter   ‹    print()›
      |                           ‹¦›
      """
      if keystroke=='\n' and not state.current_line_after_cursor:
            return
state.insert_text('\n'+state.leading_whitespace_in_current_line)
```

```
@engine.add_rule
def increment_decrement(state,keystroke):
      """
      Adds the ++ and -- operators to python
      |   ‹x›   + +   ‹x+=1›
      ...
      |   ‹x›   - -   ‹x-=1›

      Works with array_augmented_assignment
      |   ‹a[x¦]›   + +   ‹a[x]+=1¦›

      Integration test
      |   a [ ; - -   ‹a[:]-=1›
      """

      if keystroke in '+-' and state.char_before_cursor==keystroke:
            return engine.process(state,'=','1')
```

These docstrings provide documenation for each function, as well as unit tests to be performed. Here are the resulting unit test outputs for these functions:

```
RULE #42: preserve_indent

      Hitting enter can preserve the indent
      |   ‹def f():›               ‹def f():›
      |   ‹    print()¦›   enter   ‹    print()›
      |                            ‹    ¦›
            Used Rule: preserve_indent              ‹def f():↲
print()↲    ¦›
      GOOD: PASSED

      SHOULD_FAIL! Hitting enter doesn't simply go to the beginning of
the line with this rule
      |   ‹def f():›               ‹def f():›
      |   ‹    print()¦›   enter   ‹    print()›
      |                            ‹¦›
            Used Rule: preserve_indent              ‹def f():↲
print()↲    ¦›
      GOOD: FAILED

RULE #17: increment_decrement

      Adds the ++ and -- operators to python
      |   ‹x›   + +   ‹x+=1›
            Used Rule: insert_character             ‹x+¦›
            Used Rule: increment_decrement          ‹x+=1¦›
      GOOD: PASSED
      ...
      |   ‹x›   - -   ‹x-=1›
            Used Rule: insert_character             ‹x-¦›
            Used Rule: increment_decrement          ‹x-=1¦›
      GOOD: PASSED

      Works with array_augmented_assignment
```

```
|   ‹a[x¦]›   + +   ‹a[x]+=1¦›
      Used Rule: insert_character             ‹a[x+¦]›
      Used Rule: increment_decrement          ‹a[x]+=1¦›
GOOD: PASSED

Integration test
|   a [ ; - -   ‹a[:]-=1›
      Used Rule: insert_character             ‹a¦›
      Used Rule: add_matching_bracket         ‹a[¦]›
      Used Rule: semicolon_to_colon_slices    ‹a[:¦]›
      Used Rule: insert_character             ‹a[:-¦]›
      Used Rule: increment_decrement          ‹a[:]-=1¦›
GOOD: PASSED
```

Note how these unit tests show their work: for each keystroke, you'll see a line starting with 'Used Rule:', followed by the resulting state. This shows you how the string progresses as keys are pressed. Note that new lines are represented with the '↲' character in order to keep the output neat and tidy.

I encourage you to read through the other completion rules in my code (look for docstrings in functions), and to play around with the resulting program! Run it in a terminal; you can understand how these completions help firsthand.

Part 5: References

This project is almost entirely based on things that I invented. I tried to find things that were similar to my project, but I believe this project is the only one of its kind. The only thing I can think of citing would be the main library I used to write my codebase. In particular, this would be the prompt_toolkit library, which is a terminal UI framework that rp and this program both use.

@misc{
	author = {Jonathan Slenders},
	title = {Python Prompt Toolkit},
	year = {2021},
	publisher = {GitHub},
	journal = {GitHub repository},
	howpublished = {\url{https://github.com/prompt-toolkit/python-prompt-toolkit}},
}