

# SIX: ADAPTING TEXT TO CONTEXT

We've talked about how to generate text, how to fill in blanks in the text, and how to use inference and search to select what items to use to fill in blanks. But so far, when it decides to print something, it always prints it the same way. Now let's look at how we can use context information to adjust how a given item gets printed.

Even simple things like printing someone's name can be complicated. When the narrator first introduces Diana Ratcliffe, titan of industry, we'd probably use her full name:

Diana Ratcliffe, Founder and CEO of Ratcliffe Industries, surveyed the New York skyline.

On the other hand, when a character mentions her, they likely won't use her full name:

"Here are the reports you asked for, Ms. Ratcliffe," said Benjamin, her trusted assistant.

The elderly woman gazed at gift. "Oh, Diana, you shouldn't have!"

"Ratcliffe is up to something. I don't know what it is, but she now owns 51% of Algernon Pharmaceuticals."

Others might not use her name at all. If they were intimate with her, they might use an endearment such as "Mommy" or "Darling". If she had a butler, they might address her as "Madam". In an institutional context, someone might use a title, such as "Colonel", if she was in the National Guard or Army Reserves, or "Your Excellency," if she was a US ambassador. Or "Ojou-sama" if she was an anime character who was secretly the daughter of a yakuza boss.

## CUSTOMIZING OUTPUT USING MENTION

Let's say we write a task called Greet that says hello to someone:

```
Greet ?who: Hello, ?who.
```

Now, if we run `[Greet bill]`, it will print "Hello, bill." That's fine, except we'd kind of like it to capitalize Bill. We can't just say `[Greet Bill]`, because within `[ ]` code, capitalized words are names of tasks, and Step will complain there's no task named Bill.<sup>1</sup> So in the code, we use `bill` to refer to Bill the character. But we'd like some way to tell the system to print `bill` as "Bill". Enter the `Mention` task.

When you tell the system to print the value of a variable by putting its name inside the text, like this:

```
Greet ?who: Hello, ?who.
```

That's actually a shorthand for a call to a task called `Mention`. So Step treats it as a shorthand for:

```
Greet ?who: Hello, [Mention ?who].
```

---

<sup>1</sup> Technically, capitalized names are things called global variables, most of which happen to be tasks. We'll talk more about this shortly.

If you don't give the system a definition for `Mention`, it just treats it as a synonym for `Write`, a task that just prints its parameter verbatim. But you can provide your own definition of `Mention` to customize its behavior. For example, we can say:

```
Mention bill: Bill
Mention ?anythingElse: [Write ?anythingElse]
```

This says that when we `Mention` something, we should just `Write` it, unless it happens to be `bill`, in which case it should capitalize it. We can customize this however we like. For example,

```
Mention bill: Bill
Mention diana: Diana Ratcliffe, titan of industry
Mention ?anythingElse: [Write ?anythingElse]
```

Now, if we run `[Greet diana]`, it will say "Hello, Diana Ratcliffe, titan of industry." In fact, any time we print `diana`, it will print "Diana Ratcliffe, titan of industry," which will get old after a while.

## TEACHING THE SYSTEM ABOUT YOUR CHARACTERS

It's usually a good thing to start by telling the system something about your characters. You might want to tell it that they're characters, for one thing. But you might also want to tell it about their names, ages, jobs, etc. Here's an example:

```
Character diana.
Age diana: 34
Occupation diana: titan of industry
GivenName diana: Diana
FamilyName diana: Ratcliffe
```

```
Character bill.
Age bill: 26
Occupation bill: plumber
GivenName bill: Bill
FamilyName bill: Holmquist
```

Now we can define `Mention` to give first and last names for characters:

```
Mention ?c: [Character ?c] [GivenName ?c] [FamilyName ?c]
Mention ?anythingElse: [Write ?anythingElse]
```

Now any time a character is `Mentioned`, it will give their first and last names, and for anything else, it will just print it.

This isn't critical for you to learn, but there are some shorthands you can use to simplify the first method and make it more readable. If you say:

```
Mention (Character ?c): [GivenName ?c] [FamilyName ?c]
```

The computer understands that you mean this is a method that only applies when `?c` is a Character. And you can simplify the part after the colon by saying:<sup>2</sup>

```
Mention (Character ?c): ?c/GivenName ?c/FamilyName
```

If you say `?var/Task`, it just treats it as a shorthand for `[Task ?var]`. You can even simplify it further to:

```
Mention (Character ?c): ?c/GivenName+FamilyName
```

*Step* treat `?var/Task1+Task2` as a shorthand for `[Task1 ?var] [Task2 ?var]`. *Step* treats all these shorthands as identical to the original, longer version. But the intention is that it will make the code look a little less like code and a little more like text. Your mileage may vary, so use whichever form looks best to you; *Step* doesn't care.

We can also use this outside of *Mention* to try to customize how we generate text. If we write:

```
Introduce ?who:  
?who is ?who/Age years old. ?who is a ?who/Occupation.  
[end]
```

And run `[Introduce diana]`, we get “Diana Ratcliffe is 34 years old. Diana Ratcliffe is a titan of industry.” The second use of her name here feels clunky. We can make it a little better by changing it to use the character's first name:

```
Introduce ?who:  
?who is ?who/Age years old. ?who/GivenName is a ?who/Occupation.  
[end]
```

Now when we run `[Introduce diana]`, we get “Diana Ratcliffe is 34 years old. Diana is a titan of industry.”

Of course, the most fluent thing to do here would be to use a pronoun. That requires remembering what topics have been talked about recently, and that requires introducing a new *Step* feature.

## GLOBAL VARIABLES

There are really two kinds of variables in *Step*. The variables we've used so far, which start with `?`, are called *local variables* because they're specific to the particular method they appear in. You can have three different methods that all have a local variable named `?who`, but those are all different variables, generally with different values, that just happen to have the same name.

Anything that starts with a capital letter is a *global variable*. Globals are shared between all the methods. The variable `X` in one method is the same variable with the same value when referred to in another method/task. Task names are actually global variables. They just happen to be global variables whose values happen to contain code you can run.

---

<sup>2</sup> In the next reading, we'll see another use of brackets like this to make data objects called *tuples*. This use of brackets to represent a restriction on a method only applies when it appears in the parameters of a method and it looks like a single-parameter call to a predicate. Any other use is a tuple, which we'll talk about in the next reading.

But global variables don't have to store tasks. You can put whatever information into them that you want. To store a value into a global variable, you say `[set Variable = value]`. After you run that, the variable will have that new value. But if the system backtracks, it will undo the `set` operation, returning the variable to its old value.

## TRACKING DISCOURSE CONTEXT

Let's use a global variable to keep track of the most recently discussed character. We'll just have `mention` remember the most recently discussed character in the variable `They`.

```
Mention They: they
Mention (Character ?c): ?c/GivenName+FamilyName [set They = ?c]
Mention ?anythingElse: [Write ?anythingElse]
initially: [set They = nobody]
```

The first line here says that if you call `Mention` on whoever is the current value of the variable `They`, just print the pronoun "they". If the character being mentioned isn't the character in the variable `They`, then we move on to the next line, which says if they're a character, print their first and last names, and then update `They` to be the character we just mentioned. Now, if we say:

```
Introduce ?who:
?who is ?who/Age years old. ?who are a ?who/Occupation.
[end]
```

And run `[Introduce diana]`, it will print "Diana Ratcliffe is 34 years old. They are a titan of industry."

Unfortunately, we've now assumed that Diana's preferred pronoun is they. A better approach would be to let the author specify preferred pronouns for the different characters. So something like this:

```
Mention They: [PreferredPronoun They ?pronoun] [Write ?pronoun]
Mention (Character ?c): ?c/GivenName+FamilyName [set They = ?c]
Mention ?anythingElse: [Write ?anythingElse]
initially: [set They = nobody]
```

```
PreferredPronoun diana she.
PreferredPronoun bill: they.
```

This says that when we mention `They`, we just generate that character's preferred pronoun, whatever it might be. Now it will generate "she" for Diana and "they" for Bill.

## SUBJECT-VERB AGREEMENT

Now we have a new problem, though. When we run `[Introduce diana]`, it will print "Diana Ratcliffe is 34 years old. She are a titan of industry." We don't want "are" there, we want "is". But Bill goes by "they" so we still want to be able to generate "are" when talking about them. So we need to keep track of some more context information. In particular, we need to track whether the subject is in third person singular (he/she/it/Diana) or third person plural (they).

To do this, we just change the definition of `Mention` a little:

**Mention They:**

```
[PreferredPronoun They ?pronoun] [Write ?pronoun] [SetInflection ?pronoun]
[end]
```

**Mention (Character ?c):**

```
?c/GivenName+FamilyName [set They = ?c] [set ThirdPersonSingular = true]
[end]
```

```
Mention ?anythingElse: [Write ?anythingElse] [set ThirdPersonSingular = true]
```

```
SetInflection they: [set ThirdPersonSingular = false]
```

```
SetInflection ?: [set ThirdPersonSingular = true]
```

This will set `ThirdPersonSingular` to true any time we Mention something *unless* we end up using the pronoun “they”. Now we just write a task, `Is`, to print either “is” or “are” depending on what inflection we need:

```
Is: [ThirdPersonSingular] is
```

```
Is: are
```

This says that when running `Is`, we print “is” for third person singular, and “are” otherwise. This ignores a bunch of other things like first person, but we’ll leave that for another day.

Finally, we change our text from always saying “is” to instead calling the smart task `Is`:

**Introduce ?who:**

```
?who is ?who/Age years old. ?who [Is] a ?who/Occupation.
[end]
```

Now this will generate the right text for any character, regardless of pronouns.

## SUBJECTS AND OBJECTS

There’s one final wrinkle to worry about. Most Indo-European languages have a sophisticated case system. Nouns are inflected differently depending on whether they’re the subject or object of a verb, for example. That case system is almost entirely gone in English, but it lives on in our pronouns: we say “he” and “she” for subjects of verbs, but “him” and “her” for objects. So if we say something like:

```
Text ?who: ?who decided to run a marathon. It almost killed ?who.
```

And then run `[Text bill]`, we’ll get the output “Bill Holmquist decided to run a marathon. It almost killed they.” That’s dysfluent because it should be “It almost killed **them**.” We need some way of communicating the that first `?who` is the subject of a verb but the second one is the object. We can do this just by making a different task that’s like `Mention`, but generates pronouns in object case. Let’s call this task `Object`:

```
Object They: [PreferredObjectPronoun They ?pronoun] [Write ?pronoun]
```

```
Object (Character ?c): ?c/GivenName+FamilyName [set They = ?c]
```

```
Object ?anythingElse: [Write ?anythingElse]
```

Notice that this one’s simpler because it doesn’t have to keep track of whether something is third person singular, since we only worry about that for the subjects of sentences. It does need to get a different pronoun, but we can determine the object pronoun to use from the subject pronoun:

**PreferredObjectPronoun ?who them:** [PreferredPronoun ?who they]

**PreferredObjectPronoun ?who her:** [PreferredPronoun ?who she]

**PreferredObjectPronoun ?who him:** [PreferredPronoun ?who he]

This says to use “them” if they use “they”, “her” if they use “she”, etc. Now we just change our definition above to:

**Text ?who:** ?who decided to run a marathon. It almost killed [Object ?who].

Or equivalently, but more compactly:

**Text ?who:** ?who decided to run a marathon. It almost killed ?who/Object.

## HOW YOU'D REALLY DO IT

This is the basic idea of how you adapt text based on context. A real, complete version of Mention would be more complicated. Here are some things you might want in an “industrial strength” version of Mention:

- In addition to subject and object case, there are also reflexive (himself/herself) and possessive (his/her) pronouns. So you'd have tasks like Object that generated text in each of those cases.
- This version only keeps track of one person being discussed, They, and each time a new person is mentioned, it replaces the old person. That means that if it was generating text talking about Bill and Diana, it would only use a pronoun when it referred to the same person twice in a row. But since Bill and Diana use different pronouns, you can always use “they” for Bill and “she” for Diana, once you've introduced the characters. A better version would have separate variables to track the current He and She characters, as well as They. That way, having mentioned Bill recently won't prevent us from using she to refer to Diana.
- The very first time we refer to a character, we might use their full name. But subsequent times, we might just use their first name, even if we can't use a pronoun. It would read strangely to repeatedly use the full name of a character.

These issues get handled the same way that the other issues get handled: using a combination of pattern matching and using global variables to track state information.

## REAL HUMAN LANGUAGE USE

Real people are much smarter than *Step* programs and use context in much more sophisticated ways. Real people can be more nuanced. Our *Step* program just uses recency to decide on pronouns; “he” always means the most recently mentioned person, or at least the most recently mentioned person who uses male pronouns. So for example, if it generates a sentence that looks like this:

Bill *something* Ted; he *something* he *something* him

It will intend all the pronouns to refer to Ted, since Ted is the most recently mentioned person.

But humans are way smarter than that. If you take an example like:

Bill hated Ted; he thought he'd stolen his girl.

We, as human readers, can use context to understand that only the second pronoun refers to Ted; the other two refer to Bill. In principle, you could write a *Step* program that could understand these kinds of subtleties. But in practice it would be very difficult, and you'd likely want to use more sophisticated techniques than we've shown here. This is one of the many reasons AI is really hard.

Fortunately, *Step* programs aren't trying to do full natural language generation, much less natural language understanding. *Step* is designed to let humans hand-author text and then make simple changes and substitutions to it without destroying fluency.