Introduction to SAS

Welcome to the "Introduction to SAS" module, provided by the STEM Fellowship.

The SAS language is a powerful tool for statistical analysis, and in many cases, is an industry gold-standard used by business intelligence professionals the world over. If you've made it this far, you've successfully installed the Anaconda Data Science distribution, the SAS University Edition virtual machine, and the Jupyter Python Kernel bringing the interactive nature of Jupyter to the SAS space.

In this module, we will learn the basics of the SAS language. It is assumed that you have at least some background in programming, as basic computer science concepts like variables and functions will not be explained.

Our First SAS Program

To start off, we're going to show off a small program in its entirety. Look it over to get a feel for the language, then try running it to see the output. Traditionally, 'Hello, World' is the first program that is taught to new learners of any language, but in this case we'll be starting with something a little more substantive:

```
TITLE "My First SAS Program";

DATA demographics;
    INPUT gender age height weight;
    DATALINES;
M 170 21 165
F 150 19 .
F 155 .  115
M 185 22 185
F .   26  .
;
RUN;

/* Let's run something! */
PROC PRINT data=demographics;
    var height age;
run;
```

My First SAS Program

Obs	height	age
1	21	170
2	19	150
3	.	155
4	22	185
5	26	.

In this little program, we've simply declared some variables of interest, provided some data, and asked the program to print out a subset of out dataset.

We can observe a few rules that are common to all SAS programs.

- All statements must end with a semi-colon (if you're more familiar with python, many of your first errors will be from forgetting to terminate with a semicolon - be vigilant!).
- We can represent missing data with a `.` character.
- SAS statements usually start with a SAS keyword.
    - In our program we saw keywords like `TITLE`, `DATA`, `RUN`, `INPUT`, `PROC`, `PRINT` - can you guess what they do?
- Comments are C-style. Always be sure to properly document your code... future-you will thank you in six months!
- SAS programs can be as free-form as we like, so long as we remember our terminating semicolon.
    - Look at the code block under `DATALINES`. We formatted the data like we would see in a spreadsheet so it's easier for us to parse, but SAS doesn't care about line breaks! Our semicolon comes after our data, and as far as SAS is concerned, that's just one continuous line.
- Look at my last line; I forgot to capitalize `RUN`. All SAS keywords are not case-sensitive, so `RUN;` is just as valid SAS code as `run;`. We recommend that you use **ALL-CAPS** for keywords to make them easier to see, but the choice is yours. At the very least be consistent with your style - have all caps or lowercase, not both.

Names

Very briefly, let's talk about variable names in SAS.

All SAS variables must contain between 1 and 32 characters - keep them short and to the point. If your variable name hits the 32 character limit you should re-evaluate your naming scheme. The first letter of SAS variables must be a letter (`A-Z`, `a-z`) or an underscore `_` - subsequent letters need to be either a letter, an underscore, or a number.

SAS names cannot start with a number, but can contain them somewhere else in the name. As well, any other character like `$`, `%`, or `&` are allowed.

Anatomy of a SAS Program

The two main blocks that every SAS program has are a `PROC` and a `DATA` step.
Here are some rules:

- Any portion of a SAS program that begins with a `DATA` statement and ends with either a RUN statement, a `DATA` statement, or a PROC statement is called a `DATA` step.
- Any portion of a SAS program that begins with a `PROC` statement and ends with a RUN statement, a `DATA` statement, or another PROC statement is called a `PROC` step.

If you haven't guessed by now, `PROC` stands for procedure.

You use `DATA` steps to do data management. In our little program, we have a `DATA` block that we used to define our variables and enter values for them. PROC steps are used to run analysis over the data inside a `DATA` block. We would run statistics or generate a report from a `PROC` step, for example.

Finally, when we're done defining things for SAS to do, we call the `RUN` command.
NOTE: Having a terminating `RUN` command isn't strictly necessary, as SAS can infer these things, however you are highly, highly recommended to use `RUN` to delimit your blocks, as it's a good habit for clean, clear, and readable SAS code.

Variable Types

In this section, we'll look at the ways to input data into SAS, including instream data and external data.

Instream Data

Let's look at this SAS program:

```
DATA instream_dataset;
    INPUT subject_number gender $ height weight;
    DATALINES;
1000 F 150 115
1001 M 175 165
1002 F 155 125
1003 F 145 105
1004 M 180 175
1005 M 170 155
1006 M 190 200
1007 F 155 130
;
RUN;

/* let's look at our whole dataset */
PROC PRINT data=instream_dataset;
    title 'My Dataset';
RUN;

/* let's see gender versus weight */
PROC PRINT data=instream_dataset;
```

```
    title 'My Dataset - Subset';
    var gender weight;
RUN;
```

My Dataset

Obs	subject_number	gender	height	weight
1	1000	F	150	115
2	1001	M	175	165
3	1002	F	155	125
4	1003	F	145	105
5	1004	M	180	175
6	1005	M	170	155
7	1006	M	190	200
8	1007	F	155	130

My Dataset - Subset

Obs	gender	weight
1	F	115
2	M	165
3	F	125
4	F	105
5	M	175
6	M	155
7	M	200
8	F	130

We created an instream dataset.

We enclose our instream dataset inside of a `DATA-RUN` block. Notice too, that we can give our dataset a name by typing the name just after the initial `DATA`. Let's look inside the `DATA-RUN` block now.
The `DATALINES` keyword tells SAS that you want to give it instream data. This statement must be the last step in a `DATA` step and must be immediately before the data lines themselves. Only one `DATALINES` can appear in a given `DATA` step. We have a strange `$` character in our `INPUT` line here - that is a flag for SAS to consider that variable (gender) as a character variable, not a numeric (the default). Try running that cell with the `$` removed to see what happens.

Look at our second `PROC` block, and compare it to our first. Using the var keyword we can subset our dataset to take a closer look at just the variables we want to see.

External Data

For the next section, copy-paste the following into a .txt file called data.txt inside the **same** directory as your SAS code. If you are using SAS in the browser, you should create the input file on your desktop and save it. Then,

inside SAS OnDemand you should see the *Server Files and Folders* menu by default - click on the `Upload` button and point SAS to the same file you just saved.

```
toronto  2.809 -10 20
new york 8.535 -3  29
chicago  2.705 -5  23
ottawa   0.947 -10 21
calgary  1.266 -7  17
```

Make sure that the name of your textfile matches what you put after the `INFILE` command.

Note: If you are using SAS OnDemand, you have an additional step. After uploading your file, you should see it under the *Files (Home)* subfolder on the left-hand side. Right-click your input file and from the dropdown select *Properties*. You will see an entry called *Location*, and an accompanying filepath. It will look something like this: `/home/yourname/data.txt`. Copy this filepath to your clipboard. In the code below, you need to replace `data.txt` with the filepath you just copied. Therefore, you should write `INFILE '/home/yourname/data.txt'`

Now run the following SAS program:

```
DATA temperatures;
    INFILE 'data.txt';
    INPUT city $ population avg_low avg_high;
RUN;

PROC PRINT data=temperatures;
    title 'Dataset from an External File';
RUN;
```

Dataset from an External File

Obs	city	population	avg_low	avg_high
1	toronto	2.809	-10.000	20
2	new	.	8.535	-3
3	chicago	2.705	-5.000	23
4	ottawa	0.947	-10.000	21
5	calgary	1.266	-7.000	17

Everything looks good, but what happened to "new york"? Can you guess what the problem is? Try this code instead:

```
DATA temperatures;
    INFILE 'data.txt';
    INPUT city $ 1-8 population avg_low avg_high;
RUN;
```

```
PROC PRINT data=temperatures;
    title 'Dataset from an External File';
RUN;
```

Dataset from an External File

Obs	city	population	avg_low	avg_high
1	toronto	2.809	-10	20
2	new york	8.535	-3	29
3	chicago	2.705	-5	23
4	ottawa	0.947	-10	21
5	calgary	1.266	-7	17

That's better. We can tell SAS explicitly where it should look for variables. By default, it delineates different variables based on whitespace. In the case of "new york" it didn't know that it should consider the whitespace between "new" and "york" as part of the name. We have to tell SAS that it should take every character in columns 1 to 8 and stick that, whitespace and all, into `city`. If we count, we can see that `'new york'` has exactly eight characters in it. SAS is also intelligent enough to drop whitespace after a character, so we don't have `'ottawa '` as an entry. This notation is called column input, and we can explicitly map out the entire dataset if we wanted to.

Remember the 8 character limit for string variables we talked about earlier? With column input you can make strings as long as you want - for example you could write `INPUT long_name $ 1-75` to create a variable that can hold strings of up to 75 characters in length.

Other Input Types

List Input

Until now, we have had very carefully separated data that we've formatted into a tabular form. What would we do if the previous example file had contained the following?

```
toronto 2.809 -10 20
new york 8.535 -3 29
chicago 2.705 -5 23
ottawa 0.947 -10 21
calgary 1.266 -7 17
```

This is easier to type in, but we need to don some extra prep-work before we can use it correctly. We need to ensure that:

- fields are separate by some kind of delimiter (whitespace, tab, comma)
- fields are arranged in a left to right order
- missing numeric values are a `.` character
- strings cannot have embedded blanks (i.e. write `new_york` instead of `new york`)

Note, with list input any character string over 8 characters is truncated to fit.

Instead of the `DATALINES` instream keyword, we need to use the `CARDS` keyword. Let's try out the following:

```
DATA temperatures;
    INPUT city $ population avg_low avg_high;
    CARDS;
toronto 2.809 -10 20
new_york 8.535 -3 29
chicago 2.705 -5 23
ottawa 0.947 -10 21
calgary 1.266 -7 17
;
RUN;

PROC PRINT data=temperatures;
    title 'List Input Dataset';
RUN;
```

List Input Dataset

Obs	city	population	avg_low	avg_high
1	toronto	2.809	-10	20
2	new_york	8.535	-3	29
3	chicago	2.705	-5	23
4	ottawa	0.947	-10	21
5	calgary	1.266	-7	17

We can change the delimiter in a single line:

```
DATA temperatures;
    /* let's use a csv format instead */
    INFILE cards delimiter=',';

    INPUT city $ population avg_low avg_high;
    CARDS;
toronto,2.809,-10,20
new_york,8.535,.,29
chicago,2.705,-5,.
ottawa,0.947,-10,21
calgary,1.266,.,17
;
RUN;

PROC PRINT data=temperatures;
    title 'List Input Dataset';
RUN;
```

List Input Dataset

Obs	city	population	avg_low	avg_high
1	toronto	2.809	-10	20
2	new_york	8.535	.	29
3	chicago	2.705	-5	.
4	ottawa	0.947	-10	21
5	calgary	1.266	.	17

Formatting Output

We can format output, as mentioned earlier. Let's look at the following script:

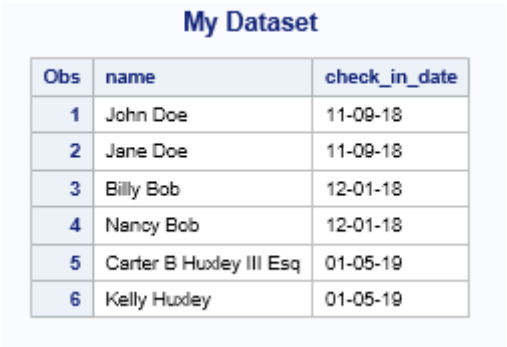
```
DATA my_dataset;
    INPUT @1 name $23.
          @25 room_number 4.
          @30 check_in_date $8.;
    DATALINES;
John Doe                 1001 11/15/18
Jane Doe                 1001 11/15/18
Billy Bob                1002 12/01/18
Nancy Bob                1002 12/01/18
Carter B Huxley III Esq 1003 01/05/19
Kelly Huxley             1004 01/05/19
;
RUN;

PROC PRINT data=my_dataset;
    title 'My Dataset';
RUN;
```

My Dataset

Obs	name	room_number	check_in_date
1	John Doe	1001	11/15/18
2	Jane Doe	1001	11/15/18
3	Billy Bob	1002	12/01/18
4	Nancy Bob	1002	12/01/18
5	Carter B Huxley III Esq	1003	01/05/19
6	Kelly Huxley	1004	01/05/19

This example uses the `@` pointer to specify data. Look at the first `@` pointer. `@1` means SAS will start looking in the first column for the name variable. Finally, `$23.` means SAS should keep in mind name is a string and it should read 23 columns. We can get very fine grained control, consider the following:

```
DATA my_dataset;
    INPUT @1 name $23.
          +6 check_in_date $8.;
    DATALINES;
John Doe                 1001 11-09-18
Jane Doe                 1001 11-09-18
Billy Bob                1002 12-01-18
Nancy Bob                1002 12-01-18
Carter B Huxley III Esq  1003 01-05-19
Kelly Huxley             1004 01-05-19
;
RUN;

PROC PRINT data=my_dataset;
    title 'My Dataset';
RUN;
```



My Dataset

Obs	name	check_in_date
1	John Doe	11-09-18
2	Jane Doe	11-09-18
3	Billy Bob	12-01-18
4	Nancy Bob	12-01-18
5	Carter B Huxley III Esq	01-05-19
6	Kelly Huxley	01-05-19

In this example, we have +6 meaning skip 6 columns from the last pointers end - that is, skip 5 columns from column 23 and start reading in 8 columns for `check_in_date`. There are a variety of formatting operators as well:

```
DATA my_dataset;
    INPUT @1 name $23.
          @30 check_in_date ddmmyy8.;
    DATALINES;
John Doe                 1001 11-09-18
Jane Doe                 1001 11-09-18
Billy Bob                1002 12-01-18
Nancy Bob                1002 12-01-18
Carter B Huxley III Esq  1003 01-05-19
Kelly Huxley             1004 01-05-19
;
RUN;

PROC PRINT data=my_dataset;
    title 'My Dataset - A';
    FORMAT check_in_date mmddyyp10.;
RUN;
```

```
PROC PRINT data=my_dataset;
    title 'My Dataset - B';
    FORMAT check_in_date mmddyyd10.;
RUN;
```

My Dataset - A

Obs	name	check_in_date
1	John Doe	09.11.2018
2	Jane Doe	09.11.2018
3	Billy Bob	01.12.2018
4	Nancy Bob	01.12.2018
5	Carter B Huxley III Esq	05.01.2019
6	Kelly Huxley	05.01.2019

My Dataset - B

Obs	name	check_in_date
1	John Doe	09-11-2018
2	Jane Doe	09-11-2018
3	Billy Bob	01-12-2018
4	Nancy Bob	01-12-2018
5	Carter B Huxley III Esq	05-01-2019
6	Kelly Huxley	05-01-2019

There are a great many different formatted input code - check the SAS official documentation for more.

Structures

Doing Calculations

Inputting data and formatting it out is well and good, but now we'll look at actually doing calculations, and some higher level topics like strings, arrays, and loops.

Let's take the example of sales (in thousands) across various stores:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
```

```
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch branch_total;
RUN;
```

Sales (in thousands)

Obs	branch	branch_total
1	downtown	590
2	uptown	505
3	dockside	245
4	east_end	405
5	west_end	95

We can also use the `WHERE` keyword in our `PRINT` procedure to select only certain values. For example, say we only want to look at branches that make less than $300,000 in a given year:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch branch_total;
    WHERE branch_total < 300;
RUN;
```

Sales (in thousands)

Obs	branch	branch_total
3	dockside	245
5	west_end	95

Say there was an accounting error, and in quarter 3 each store actually made $5000 more than reported. We can go back into our program and add 5 to each branch's value in Q3:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    sales_Q3 = sales_Q3 + 5;
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch branch_total;
RUN;
```

Sales (in thousands)

Obs	branch	branch_total
1	downtown	595
2	uptown	510
3	dockside	250
4	east_end	410
5	west_end	100

We can use the regular mathematical operations as you would expect, here's a quick list:

Symbol	Description
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
-x	negation

Let's use these to calculate the tax (13%) that each branch has to pay, and throw in the net income for good measure:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    sales_Q3 = sales_Q3 + 5;
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch branch_total tax_owed net_income;
RUN;
```

Sales (in thousands)

Obs	branch	branch_total	tax_owed	net_income
1	downtown	595	77.35	517.65
2	uptown	510	66.30	443.70
3	dockside	250	32.50	217.50
4	east_end	410	53.30	356.70
5	west_end	100	13.00	87.00

Functions

We can also call functions. Let's use the mean function to find the average gross income per location:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_mean_income = mean(sales_Q1,sales_Q2,sales_Q3,sales_Q4);
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;
```

```
PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch branch_mean_income branch_total tax_owed net_income;
RUN;
```

Sales (in thousands)

Obs	branch	branch_mean_income	branch_total	tax_owed	net_income
1	downtown	147.50	590	76.70	513.30
2	uptown	126.25	505	65.65	439.35
3	dockside	61.25	245	31.85	213.15
4	east_end	101.25	405	52.65	352.35
5	west_end	23.75	95	12.35	82.65

Here are some common functions:

Function	**Call**
INT - get value as an integer	a = int(x)
ABS - get absolute value	a = abs(x)
SQRT - get the square root	a = sqrt(x)
MIN - the minimum value	a = min(x,y,z)
MAX - the maximum value	a = max(x,y,z)
SUM - the sum	a = sum(x,y,z)
MEAN - the mean	a = mean(x,y,z)
ROUND - round to the given unit	a = round(x, 1)
LOG - the log	a = log(x)
LAG - the value of the argument in the previous position	a = lag(x)
DIF - the difference between the values of the argument in the current and previous observations	a = dif(x)
N - number of non missing values	a = n(x)
NMISS - number of missing values	a = nmiss(x)

Of course, functions are composable; i.e. `x = round(sum(a,b,c,d), 1)` is valid SAS code.

We can do operations on character variables as well. Let's rank each branch. If they made less than $100,000 we'll give them a score of **C**, if they made between $100,000 and $300,000 then we'll give a score of **B**, and if they made more than $300,000 we'll give them a score of **A**:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;

    if (net_income < 100) then score = 'C';
    else if (net_income >= 100 & net_income < 300) then score = 'B';
    else score = 'A';

    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch net_income score;
RUN;
```

Sales (in thousands)

Obs	branch	net_income	score
1	downtown	513.30	A
2	uptown	439.35	A
3	dockside	213.15	B
4	east_end	352.35	A
5	west_end	82.65	C

We've introduced a lot in the last example. We are able to use conditional if statements like in every other language. We can create and assign a new character variable by testing a condition and calling a then block. We can also create `if` ... `else if` ... `else` ... blocks as in python.

The logical and comparative operations are also very similar to python:

Symbol	Definition
=	equals

Symbol	Definition
^= or -= or ~=	not equals
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
in	equal to on of a list
&	AND
! or |	OR
^ or ~ or -	NOT

Of special note, || is the concatenation operator.

Strings

Let's make our output a bit more interesting, and include some human-readable text that we'll append our grade to. Maybe we want this for a report, or to show to a manager. We can set up a string `base` to hold the common string `The store had a performance of grade` and concatenate it with `score` using the `||` operator:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;

    base="The store had a performance of grade ";

    if (net_income < 100) then score = base || 'C';
    else if (net_income >= 100 & net_income < 300) then score = base || 'B';
    else score = base || 'A';

    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
```

```
    var branch net_income score;
RUN;
```

Sales (in thousands)

Obs	branch	net_income	score
1	downtown	513.30	The store had a performance of grade A
2	uptown	439.35	The store had a performance of grade A
3	dockside	213.15	The store had a performance of grade B
4	east_end	352.35	The store had a performance of grade A
5	west_end	82.65	The store had a performance of grade C

Arrays

Let's change our program to use arrays instead. We can make an array with the `ARRAY` keyword. We need to then provide a name and the length of the array, followed by the input values we want in the array. Our complete syntax would be `ARRAY name(len) varStart-varFinish`.

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    ARRAY sales(4) sales_Q1-sales_Q4;

    branch_mean = MEAN(OF sales(*));
    branch_total = SUM(OF sales(*));
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;

    IF (net_income < 100) THEN score = 'C';
    ELSE IF (net_income >= 100 & net_income < 300) THEN score = 'B';
    ELSE score = 'A';

    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC PRINT data=sales;
    title 'Sales (in thousands)';
    var branch branch_total branch_mean net_income score;
RUN;
```

Sales (in thousands)

Obs	branch	branch_total	branch_mean	net_income	score
1	downtown	590	147.50	513.30	A
2	uptown	505	126.25	439.35	A
3	dockside	245	61.25	213.15	B
4	east_end	405	101.25	352.35	A
5	west_end	95	23.75	82.65	C

Notice we can use functions on arrays as well to calculate the mean and sum. We can run this function over the entire array using the `OF name(*)` syntax, which returns the entire array.

We can also search an array to see if it contains a certain value. Let's try to make a SAS program to see which one of our friends are ready for St. Patrick's day. If they're wearing green they're invited to our party, otherwise they're not!

```
DATA friends;
    INPUT name $ age $ shirt_colour $;
    ARRAY friends(*) $ name age shirt_colour;

    IF 'Green' in friends THEN invite = "Yes";
    ELSE invite = "No";

    DATALINES;
George  20 Green
Sam     19 Red
Caitlin 23 Green
Kim     21 Blue
Jamie   25 Yellow
;
RUN;

PROC PRINT data=friends;
    title 'Our Friends (Green)';
    var name age shirt_colour invite;
        WHERE invite = "Yes";
RUN;
```

Our Friends (Green)

Obs	name	age	shirt_colour	invite
1	George	20	Green	Yes
3	Caitlin	23	Green	Yes

We saw some new array syntax here. We gave the array a length of `(*)`, that allows SAS to find the length itself if for whatever reason we don't want to input it ourselves. We also didn't use a range, and gave each variable specifically. This can be useful for some programs.

Loops

SAS has a variety of loop structures, we'll quickly go over the `DO Index`, `DO WHILE`, and `DO UNTIL`.

DO Index

This loop is equivalent to a `for` loop in Python. The loop continues until the stop value of the index variable. Let's use SAS to calculate the sum of every digit from 0 to 10, so our answer would be 1 + 2 + 3 + ... + 9 + 10.

```
DATA sum_digits;
sum = 0;
DO i = 1 to 10;
   sum = sum + i;
END;

PROC PRINT DATA = sum_digits;
    title 'The Sum of 1 to 10';
RUN;
```

The Sum of 1 to 10

Obs	sum	i
1	55	11

Our answer is 55, check it yourself and you'll see our loop worked! Notice that out index `i` is now 11. We declared our loop in the form `DO index = start TO finish`. After each iteration of the loop, we check if we're at the `finish` value yet. After iteration 10 our index `i` became 11, and that condition was now false (i.e. i > 10) so we broke out of our loop.

DO WHILE

This loop is equivalent to a `while` loop in Python. The loop continues until the while condition becomes `false`. Let's calulate the same thing as we did previously with a `DO WHILE` instead:

```
DATA sum_digits;
sum = 0;
i = 1;
DO WHILE(i<=10);
   sum = sum + i;
   i = i + 1;
END;
```

```
PROC PRINT DATA = sum_digits;
    title 'The Sum of 1 to 10';
RUN;
```

The Sum of 1 to 10

Obs	sum	i
1	55	11

In this case we had to declare our `i` index variable outside of the loop (which only contains our condition) and we also had to increment it ourselves too - in the previous example the `DO Index` structure handled this for us.

DO UNTIL

This loop is like a `do ... while` statement, which doesn't exist explicitly in Python but can be found in languages like C. It is similar to the `DO WHILE` loop discussed previously, but instead of the condition being checked then executing a loop iteration, it executes a loop iteration then checks the condition.

```
DATA sum_digits;
sum = 0;
i = 1;
DO UNTIL(i<=10);
   sum = sum + i;
   i = i + 1;
END;

PROC PRINT DATA = sum_digits;
    title 'The Sum of 1 to 10';
RUN;
```

The Sum of 1 to 10

Obs	sum	i
1	55	11

This example is pretty much exactly the same as the previous one. However, under the hood know that the `DO WHILE` always checks the conditional first before it executes a loop, while the `DO UNTIL` runs the body of the loop first, then checks the conditional.

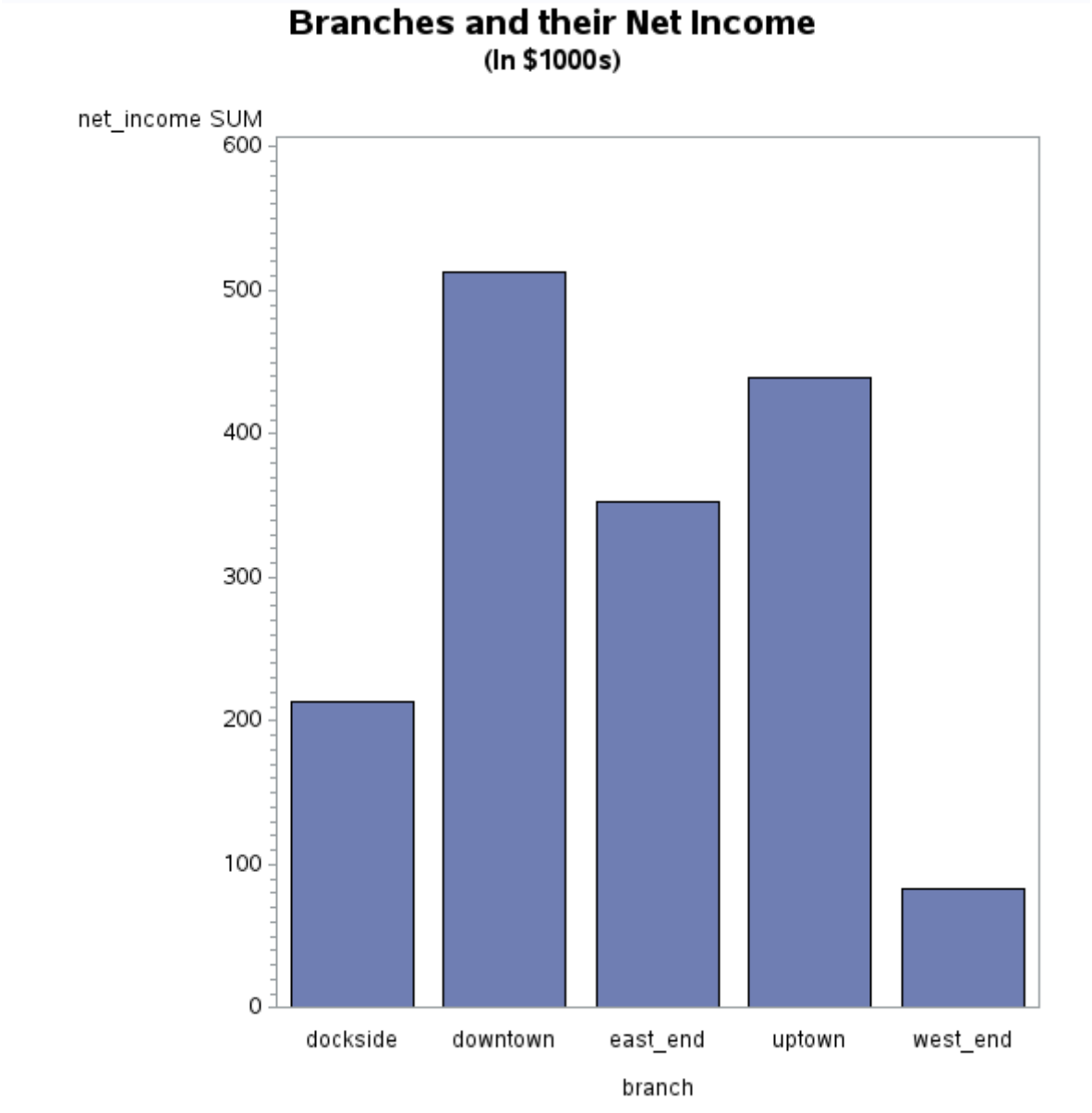
If you have a situation where the condition starts false, like (i < 0), a `DO WHILE` loop will **never** execute, and SAS will just skip over everything in the loop body like it wasn't there. On the other hand, in the same situation a `DO UNTIL` loop will execute the loop body **once** before moving on.

Graphs

We can display a variety of graphs as output in SAS. In this section we'll be going over basic charting techniques you can use to generate professional graphics.

Histograms

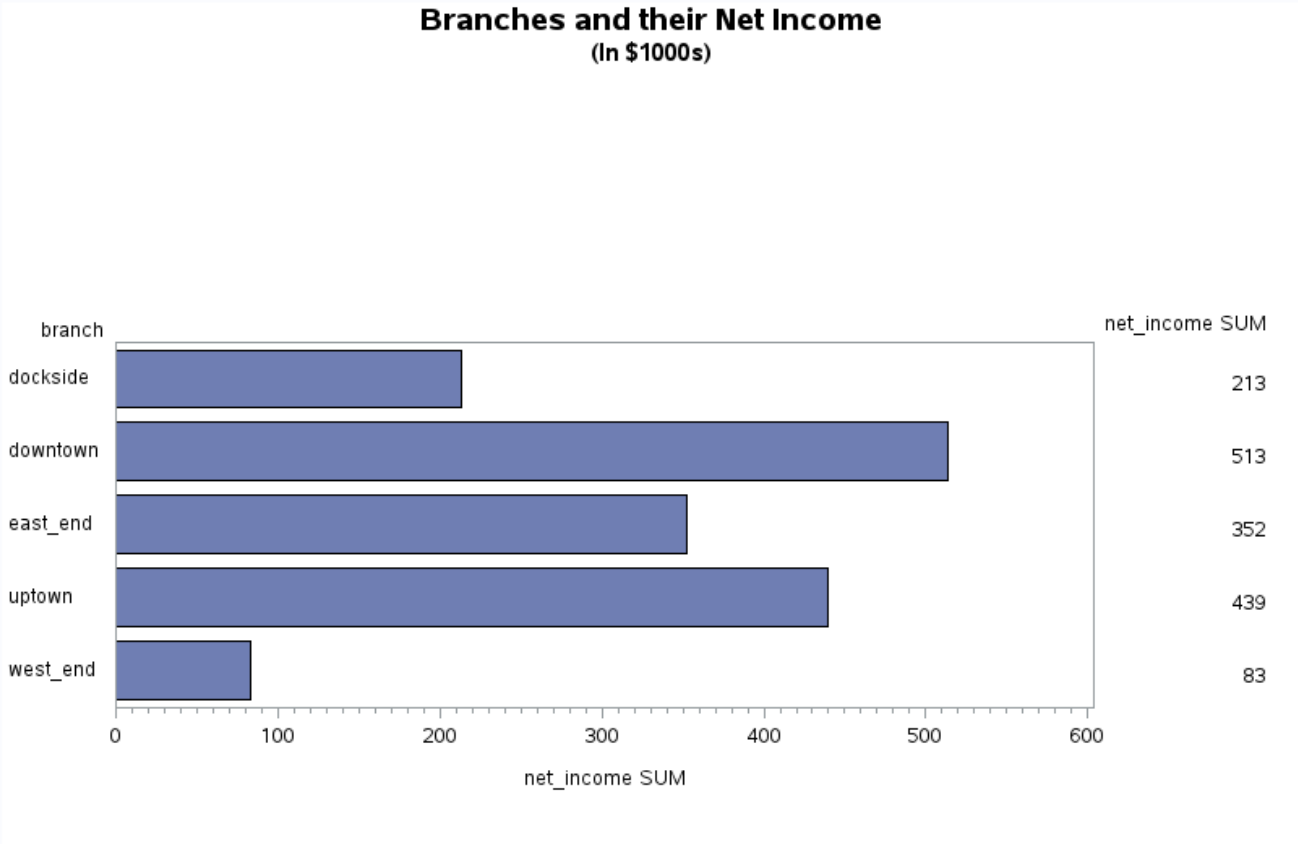
```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_mean_income = mean(sales_Q1,sales_Q2,sales_Q3,sales_Q4);
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;

PROC GCHART DATA = sales;
    title1 'Branches and their Net Income';
    title2 '(In $1000s)';
    VBAR branch / sumvar=net_income;
    RUN;
QUIT;
```



If we needed, we could make the chart horizontal too:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_mean_income = mean(sales_Q1,sales_Q2,sales_Q3,sales_Q4);
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
;
RUN;
```

```
PROC GCHART DATA = sales;
    title1 'Branches and their Net Income';
    title2 '(In $1000s)';
    VBAR branch / sumvar=net_income;
    RUN;
QUIT;
```

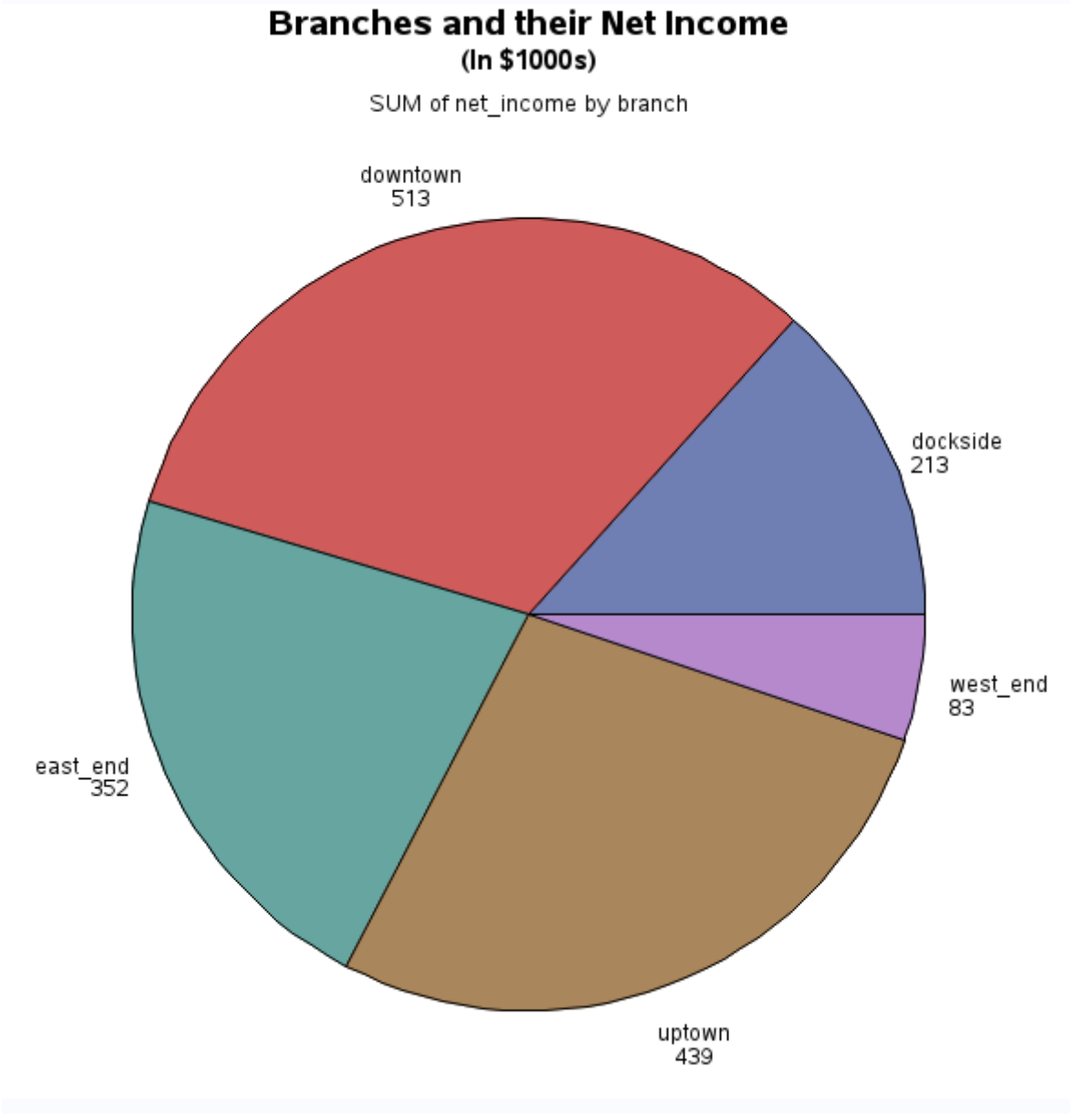


Pie Charts

It would be more useful if we could see how each branch impacts the total for the company, so let's make a pie chart:

```
DATA sales;
    INPUT branch $ sales_Q1 sales_Q2 sales_Q3 sales_Q4;
    branch_mean_income = mean(sales_Q1,sales_Q2,sales_Q3,sales_Q4);
    branch_total = sales_Q1 + sales_Q2 + sales_Q3 + sales_Q4;
    tax_owed = branch_total * 0.13;
    net_income = branch_total - tax_owed;
    DATALINES;
downtown 145 150 140 155
uptown   125 115 135 130
dockside 65  70  50  60
east_end 100 90  110 105
west_end 25  30  25  15
```

```
;
RUN;

PROC GCHART DATA = sales;
    title1 'Branches and their Net Income';
    title2 '(In $1000s)';
    PIE branch / sumvar=net_income;
    RUN;
QUIT;
```



Charts in SAS are a huge topic, covering everything is beyond the scope of the introductory tutorial. This section was just designed to give you a taste of what can be done. If you want to learn more, please consult the official SAS documentation!

Final Words and Resources

This completes the introduction to SAS module. This module was by no means exhaustive, and was designed merely to bring you to a level where you are able to read and understand SAS code that other people have written. There are hundreds more functions and visualization options that we haven't talked about, and those are left to you to explore. Please consult the official SAS documentation, the people at SAS have put a lot of time and effort into making it and their work is top-notch.

There are **hundreds** of video tutorials available at the official SAS tutorial page, and if you are interested in SAS then we strongly recommend that you give this page a look.

STEM Fellowship extends it's heartfelt gratitude to our partners at SAS for all their help. Look forward for many more SAS-based modules in upcoming Big Data Challenges.