

LING/C SC/PSYC 438/538

Lecture 5

Sandiway Fong

Today's Topics

- Homework 4 Review
- A bit more on quoting
- perlintro: scalars and arrays
- Next time: Homework 5

Homework 4 Review

Hello, my name is Inigo Mont oya . ← Mont oya

Written by Transformer · transformer.huggingface.co

*Lift a locks creen

Life is like riding a bic yle.

concrete abstract deaf

ness

Is "ness" a suffix?

He ran shoebox

-sized errands in his

es and baskets in the front

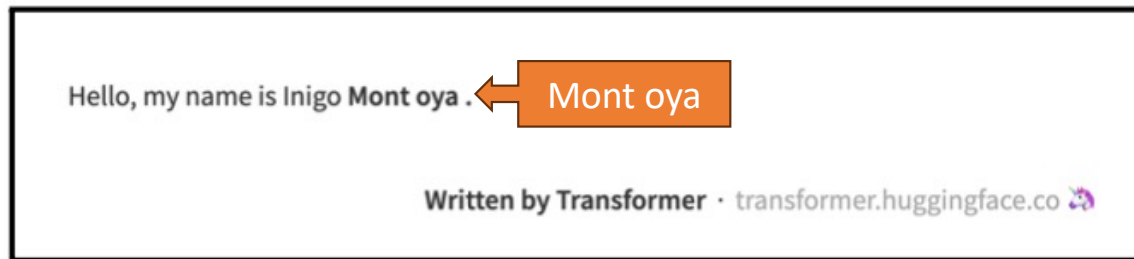
es and carried a large wooden

Homework 4 Review

- Large Language Models (LLMs) do **Sub-Word Tokenization**
 - Each token ultimately is expressed as a vector (floating point numbers)
- Example
 - *15 words becomes 20 tokens*
 - ```
>>> string = 'Balkanization is the fragmentation of a larger region or state into smaller regions or states.'
```
  - ```
{'input_ids': [101, 18903, 2734, 1110, 1103, 17906, 1891, 1104, 170, 2610, 1805, 1137, 1352, 1154, 2964, 4001, 1137, 2231, 119, 102], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```
 - ```
>>> len(encoded['input_ids'])
```
  - 20
  - ```
>>> tokenizer.decode(encoded['input_ids'][1])
```
 - 'Balkan'
 - ```
>>> tokenizer.decode(encoded['input_ids'][2])
```
  - '##ization'

# Homework 4 Review

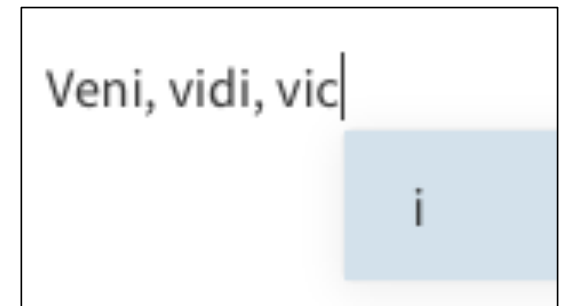
- LLMs can't know every word!



- Stingy Sub-word tokenization (*vocab. size is a problem*)
  - *big vocabulary size forces the model to have an enormous embedding matrix as the input and output layer*
  - GPT: vocab size: 40,478.
  - **GPT-2: vocab size: 50,257. bytes as (base) characters: 256. 50K merges**
  - WordPiece (BERT): merge most common bigram characters

- Someone tried:

- *"I came, I saw, I conquered"*



attributed to  
*Julius Caesar*

# Homework 4 Review

When the stars

e ha

, tha

ing

The answer is simple though. The input *in the stars* has been corrupted by introducing letters from another alphabet that look exactly the same: *When the stars*.

- GPT-2:
  - 8 bits as base characters
  - Latin-1 character set assumed
  - Therefore, UTF-8 corruption?

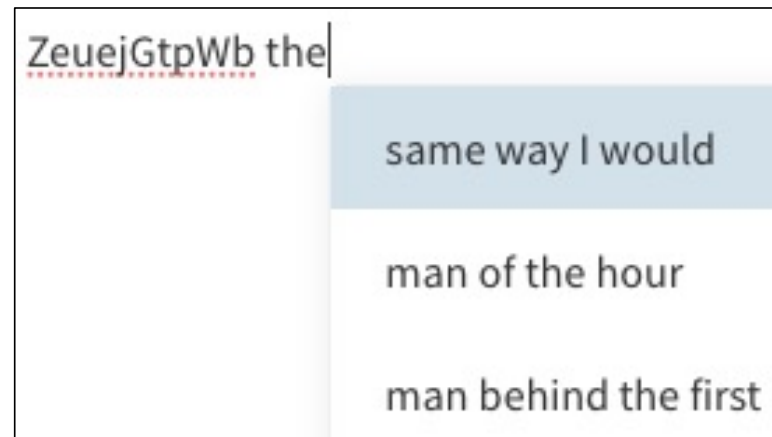
- Curse/expletive words are probably filtered out too... **er, actually no.**



*Four Weddings and a Funeral*  
(movie)

# Homework 4 Review

- It can ignore (ungrammatical) context, take the last (few) words as the new starting point



# Homework 4 Review

Language: infinite employment of finite means (von Humboldt, cited by Chomsky).

- Also "*Galileo expressed his or an amazement at what is in fact an astonishing fact with a finite number of symbols one can construct in the mind an infinite number of linguistically formulated thoughts and can even go on to reveal to others who have no access to our minds their innermost workings.*" 2023 Keio Lecture 2 (00:36) (Chomsky)
- Sod it, why not?



"Sod it, why not?" is an informal expression used to convey a casual or carefree attitude toward making a decision or taking a risk. It suggests a willingness to go ahead with something even if it might seem impulsive or unconventional. It's often used when someone wants to do something on a whim without overthinking it.



# Homework 4 Review

Sod it, why not|  
try to  
have a  
? You

A sod-it-why-not|  
-  
?

He chested the ball down, swivelled and cracked a **sod-it-why-not** shot that took a slight deflection off Evans and beat the diving Onana at the near post.  
(Guardian 9/3/2023)

# Homework 4 Review

Chomsky (1956)

grammar. There is no general relation between the frequency of a string (or its component parts) and its grammaticalness. We can see this most clearly by considering such strings as

(14) colorless green ideas sleep furiously

which is a grammatical sentence, even though it is fair to assume that no pair of its words may ever have occurred together in the past. Notice that a speaker of English will read (14) with the ordinary intonation pattern of an English sentence, while he will read the equally unfamiliar string

(15) furiously sleep ideas green colorless

with a falling intonation on each word, as in

# Homework 4 Review

- Colorless green ideas sleep furiously

Colorless green  
and black.  
or orange, but  
/yellow, with

Colorless green ideas  
.   
S  
are the most

Colorless green ideas sleep  
.   
with the sun.

Colorless green ideas sleep furiously  
.

Colorless green ideas sleep furiously about it, for some time unknown .

# Homework 4 Review

5! = 120  
permutations

```
((colorless (JJ)) (green (JJ)) (ideas (NNS)) (sleep (VBP)) (furiously (RB)) (. (.) .))
((colorless (JJ)) (green (JJ)) (ideas (NNS)) (furiously (RB)) (sleep (NN)) (. (.) .))
((colorless (JJ)) (green (JJ)) (sleep (NN)) (ideas (NNS)) (furiously (RB)) (. (.) .))
((colorless (JJ)) (green (JJ)) (sleep (NN)) (furiously (RB)) (ideas (NNS)) (. (.) .))
((colorless (JJ)) (green (JJ)) (furiously (JJ)) (ideas (NNS)) (sleep (NN)) (. (.) .))
((colorless (JJ)) (green (JJ)) (furiously (RB)) (sleep (VB)) (ideas (NNS)) (. (.) .))
((colorless (JJ)) (ideas (NNS)) (green (JJ)) (sleep (NN)) (furiously (RB)) (. (.) .))
((colorless (JJ)) (ideas (NNS)) (green (JJ)) (furiously (RB)) (sleep (NN)) (. (.) .))
((colorless (JJ)) (ideas (NNS)) (sleep (VBP)) (green (JJ)) (furiously (RB)) (. (.) .))
((colorless (JJ)) (ideas (NNS)) (sleep (VBP)) (furiously (RB)) (green (JJ)) (. (.) .))
((colorless (JJ)) (ideas (NNS)) (furiously (RB)) (green (JJ)) (sleep (NN)) (. (.) .))
((colorless (JJ)) (ideas (NNS)) (furiously (RB)) (sleep (VBP)) (green (JJ)) (. (.) .))
((colorless (JJ)) (sleep (NN)) (green (JJ)) (ideas (NNS)) (furiously (RB)) (. (.) .))
((colorless (JJ)) (sleep (NN)) (green (JJ)) (furiously (RB)) (ideas (NNS)) (. (.) .))
```

Colorless sleep

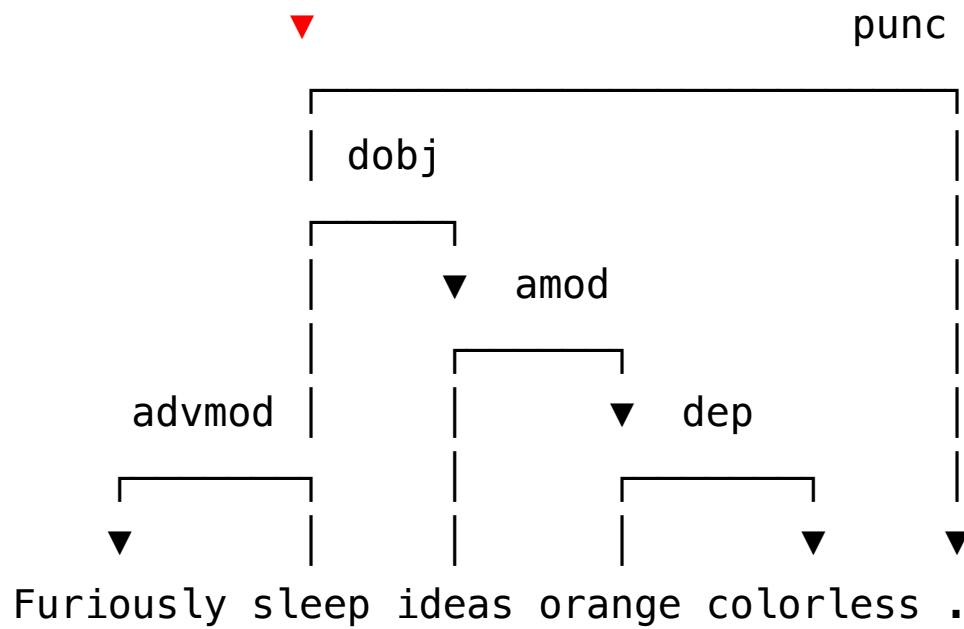
Colorless sleep green

Colorless sleep green furiously

Colorless sleep green furiously ideas

- (green), the color
- , a mind of thoughts
- A long sleep

# Homework 4 Review



**✗**  
*doesn't make  
sense either  
syntactically or  
semantically*

# Shell vs. Programming Language

From last time, a historic conflict over quoting behavior (' ").

- On the command line:
  - the Terminal (Shell) gets first dibs, and
  - the programming language, e.g. Perl, gets seconds
- **Choice:**
  - Understand the quoting rules for the Shell, or
  - Write your program always using a plain text file, e.g. `prog.perl`, run using:  
`perl prog.perl`
  - advantage: you don't have to worry about the Shell quoting rules

# Windows PowerShell and Python

```
Select Windows PowerShell
PS C:\Users\sandiway> python -c "print('hello')"
Traceback (most recent call last):
 File "<string>", line 1, in <module>
NameError: name 'hello' is not defined
PS C:\Users\sandiway> python -c 'print("hello")'
hello
PS C:\Users\sandiway> python -c "print('hello')"
Traceback (most recent call last):
 File "<string>", line 1, in <module>
NameError: name 'hello' is not defined
PS C:\Users\sandiway> python -c 'print("hello")'
Traceback (most recent call last):
 File "<string>", line 1, in <module>
NameError: name 'hello' is not defined
PS C:\Users\sandiway> python -c "print('hello')"
hello
PS C:\Users\sandiway> █
```

**Python** uses single and double quotes interchangeably to delimit strings.

- Unquoted string is a variable name (or keyword)

doubled single quotes inside single-quoted string

single quotes inside double-quoted string

# Windows PowerShell and Perl

```
Windows PowerShell
PS C:\Users\sandaway> perl -e "print 'hello'"
hello
PS C:\Users\sandaway> perl -e "print 'hello\n'"
hello\n
PS C:\Users\sandaway> $word = 'class'
PS C:\Users\sandaway> $word
class
PS C:\Users\sandaway> perl -e "print \"hello $word\n\""
hello class
PS C:\Users\sandaway> perl -e 'print "hello\n"'
hello
PS C:\Users\sandaway> perl -e "print \"hello\n\""
hello
PS C:\Users\sandaway> █
```

Perl is quirky on Windows:

- " needs to be \"
- Inside single quotes, \" is ok to Perl
- Inside double quotes, needs to be \"



# Bash Shell quoting

- Bash shell (MacOS, Linux):
  - manual: <http://www.gnu.org/software/bash/manual/>

## 3.1.2.2 Single Quotes

Enclosing characters in single quotes (‘ ’) preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

want this (@a is an array):

```
@a=('a', 'b', 'c')
```

but we can't write:

```
perl -e '@a=(\'a\',\'b\',\'c\'); print "@a\n"'
```

So what can we do? (use double quoting: *see next slide*)

1. ' ... ' *fine if no ' inside*
2. ' ... ' ... ' ... ' *doesn't work*
3. ' ... \' ... \' ... ' *cannot work*

# Bash Shell quoting

- Bash shell (MacOS, Linux):

## 3.1.2.3 Double Quotes

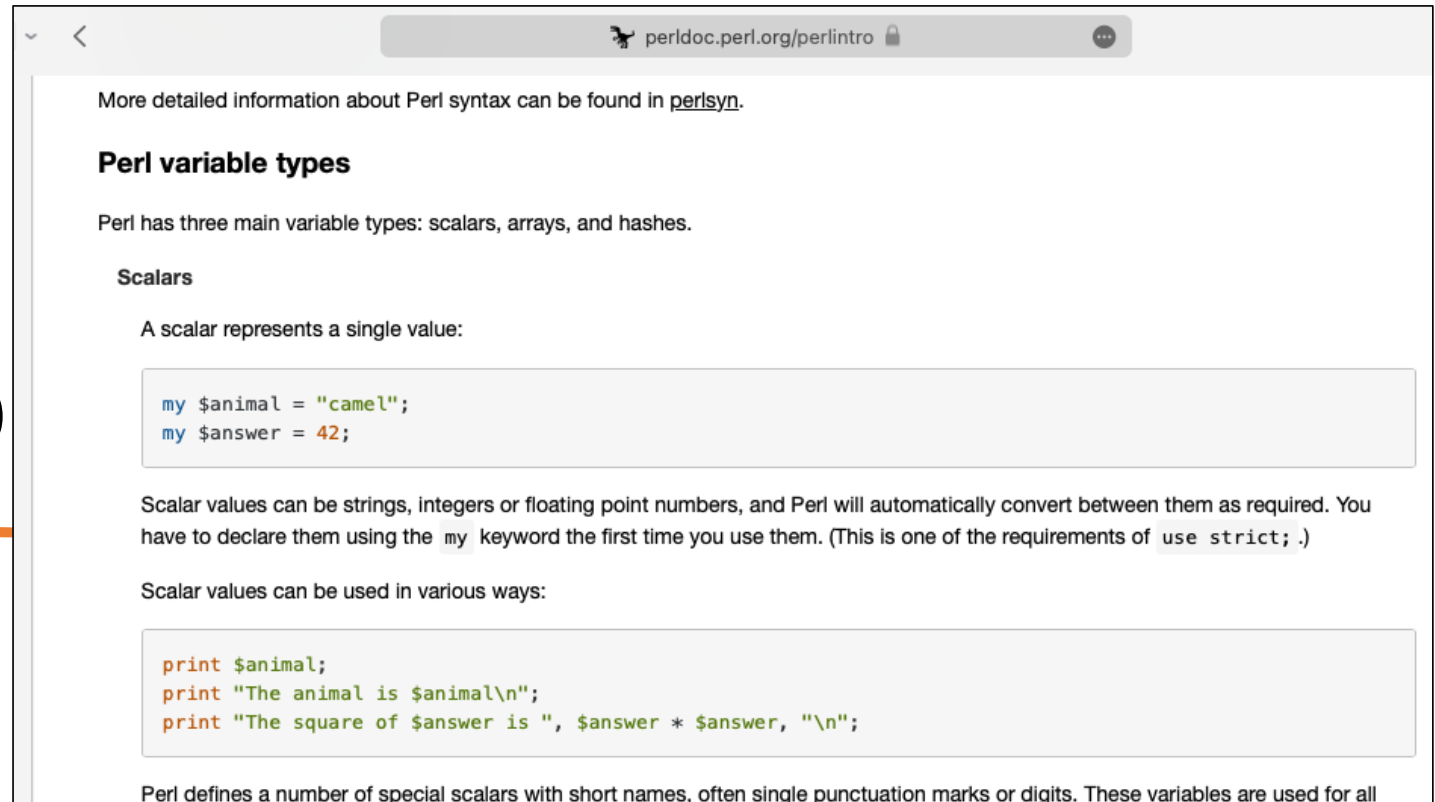
Enclosing characters in double quotes (“”) preserves the literal value of all characters within the quotes, with the exception of ‘\$’, ‘~’, ‘\’, and, when history expansion is enabled, ‘!’. When the shell is in POSIX mode (see [Bash POSIX Mode](#)), the ‘!’ has no special meaning within double quotes, even when history expansion is enabled. The characters ‘\$’ and ‘~’ retain their special meaning within double quotes (see [Shell Expansions](#)). The backslash retains its special meaning only when followed by one of the following characters: ‘\$’, ‘~’, ‘”’, ‘\’, or `newline`. Within double quotes, backslashes that are followed by one of these characters are removed. Backslashes preceding characters without a special meaning are left unmodified. A double quote may be quoted within double quotes by preceding it with a backslash. If enabled, history expansion will be performed unless an ‘!’ appearing in double quotes is escaped using a backslash. The backslash preceding the ‘!’ is not removed.

**can write \** (*but not elegant*):

```
perl -e "@a=(\"a\", \"b\", \"c\"); print \"@a\n\""
```

# perlintro

<https://perldoc.perl.org/perlintro.html>



More detailed information about Perl syntax can be found in [perlsyn](#).

## Perl variable types

Perl has three main variable types: scalars, arrays, and hashes.

### Scalars

A scalar represents a single value:

```
my $animal = "camel";
my $answer = 42;
```

Scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. You have to declare them using the `my` keyword the first time you use them. (This is one of the requirements of `use strict;`.)

Scalar values can be used in various ways:

```
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

Perl defines a number of special scalars with short names, often single punctuation marks or digits. These variables are used for all

# perlintro

- Please read the Scalars (\$) section ...

```
[Machine$ perl
$animal = "camel";
print "Selected animal is $animal\n"
Selected animal is camel
Machine$
```

control-D

Machine\$ **is my prompt, don't type that!**

- I am using the terminal as the file input to Perl
- Type **control-D** (EOF = End Of File) to send to Perl.

Windows PowerShell

```
PS C:\Users\sandiway> perl
$ans = 42;
print "$ans squared is ", $ans * $ans, "\n"
^Z
42 squared is 1764
PS C:\Users\sandiway>
```

control  
-Z

PS C:\Users\sandiway> **is my prompt, don't type that!**

- I am using the terminal as the file input to Perl
- Type **control-Z** RETURN (EOF) to send to Perl.

# perlintro

## Non-scalar data type: **array**

- prefix with **@**, array is @name (name = array name)
- indexed from 0
- \$name[index], an element of the @name array (**notice scalar \$**)
- \$#name, index of last element
- print "@name" (spaces inserted), print @name (no spaces)

controlled by system variable \$"      **default value: a space**

# perlintro

```
[Machine$ perl -e '@a=(1,2,3,4,5); print $a[-1],"\n"'
5
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[1..3]\n"'
2 3 4
[Machine$ perl -e '@a=(1,2,3,4,5); print @a[1..3],"\n"'
234
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[0,2,3]\n"'
1 3 4
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[2..]\n"'
syntax error at -e line 1, near "..]"
Execution of -e aborted due to compilation errors.
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[2..$#a]\n"'
3 4 5
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[..3]\n"'
syntax error at -e line 1, near "[.."
Execution of -e aborted due to compilation errors.
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[0..3]\n"'
1 2 3 4
```

← not in Python

← Python a[2:]

← Python a[:4]

# perlintro

- Perl

```
[Machine$ perl -e '@a=(1,2,3,4,5); print $a[-1],"\n"'
5
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[1..3]\n"'
2 3 4
[Machine$ perl -e '@a=(1,2,3,4,5); print @a[1..3],"\n"'
234
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[0,2,3]\n"'
1 3 4
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[2..]\n"'
syntax error at -e line 1, near "..]"
Execution of -e aborted due to compilation errors.
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[2..$#a]\n"'
3 4 5
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[..3]\n"'
syntax error at -e line 1, near "[..]"
Execution of -e aborted due to compilation errors.
[Machine$ perl -e '@a=(1,2,3,4,5); print "@a[0..3]\n"'
1 2 3 4
```

- Python

```
[>>> a = [1,2,3,4,5]
[>>> print(a[-1])
5
[>>> print(a[1:4])
[2, 3, 4]
[>>> print(a[2:])
[3, 4, 5]
[>>> print(a[:4])
[1, 2, 3, 4]
[>>>
```



# perlintro

## Notes from the tutorial:

- semicolon (;) is not always necessary
  - **Command separator** semantics vs. end of command (termination) token
  - **Best practice?** Terminate every command with a semicolon
- Variable types:
  - **Every variable type has its own namespace. (cf. Python)**
  - This means that \$foo and @foo are two different variables.
  - It also means that \$foo[1] is a part of @foo, not a part of \$foo. **This may seem a bit weird, but that's okay, because it is weird.**



# Perl Arrays

*like a simple ordered list...* (in **Python**, we use a list/sequence)

- Literal:
  - `@ARRAY = ( ... , ... , ... )` (round brackets; comma separator)      **Python:** `array = [ ... , ... , ... ]`
- Access:
  - `$ARRAY[ INDEX ]` (zero-indexed; negative indices ok; slices ok)      **Python:** `array[index]`
- Index of last element:
  - `$#array` (a scalar)
- Last element
  - `$array[$#array]` or `$array[-1]`      **Python:** `array[-1]`
- Slice of an array
  - `@array[i..j]` (i,j indices)      **Python:** `array[i:j]` (i/j optional also step, e.g. `::-1`)
- Coercion
  - `@ARRAY`      #size in scalar context      **Python:** `len(array)`
  - `scalar(@ARRAY)`

# Perl Arrays

- Built-in arrays:
  - @ARGV (command line arguments; coercion possible)
  - \$ARGV[0] (1<sup>st</sup> argument)
  - \$0 (program name)
  - @\_ (sub(routine) arguments)
- **Example:**

myprog.perl

```
1 print "\$0:$0\n";
2 print "$ARGV[0]\n";
3 print $ARGV[0]+$ARGV[0], "\n";
```

```
[Machine$ perl myprog.perl 1
$0:myprog.perl
1
2
Machine$ █
```

# Perl Arrays

- Python argv:
  - `import sys`
  - `sys.argv` (list of command arguments as strings)
  - `sys.argv[0]` (Python script name)
  - `sys.argv[1]` (1<sup>st</sup> argument)

- **Example:**

myprog.py

```
1 import sys
2 print("argv[0]:" + sys.argv[0])
3 print(sys.argv[1]+sys.argv[1])
```

```
[Machine$ python3 myprog.py 1
argv[0]:myprog.py
11
Machine$
```

`int(sys.argv[1])` to convert string into an integer

# Perl Arrays

```
1 print "\$0:$0\n";
2 print "$ARGV[0]\n";
3 print $ARGV[0]+$ARGV[0], "\n";
4 print $ARGV[0].$ARGV[0], "\n";
```

```
[Machine$ perl myprog.perl 1
$0:myprog.perl
1
2
11
Machine$
```

# Perl Arrays

- Built-in functions:
  - `sort @ARRAY; reverse @ARRAY;`
  - `push @ARRAY, $ELEMENT; pop @ARRAY;` (operates at right end of array)
  - `shift @ARRAY; unshift @ARRAY, $ELEMENT,` (left end of array)
  - `splice @ARRAY, $OFFSET, $LENGTH, $ELEMENT`
    - *\$ELEMENT above can be @ARRAY*
- **Python:**
  - `sorted(array)` (new array), `array.sort()` (modify *array*), `array.reverse()`
  - **NO push** (use `array.append()` instead), `array.pop()`,
  - **No shift/unshift** etc... (but can use slicing and concatenation)

# perlintro: Perl Arrays

- **shift ARRAY**            Similar to pop/push, but operates at the left end of the array
- **shift**  
Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the `@_` array within
- **unshift ARRAY,LIST**  
Does the opposite of a shift. Or the opposite of a push, depending on how you look at it. Prepends list to the front of the array and returns the new number of elements in the array.

Python doesn't have these defined but can be simulated via slicing and concatenation:

```
array[1:]
list + array
```

# perlintro: Perl Arrays

- **splice ARRAY,OFFSET,LENGTH,LIST**

- **splice ARRAY,OFFSET,LENGTH**
- **splice ARRAY,OFFSET**
- **splice ARRAY**

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or `undef` if no elements are removed. The array grows or shrinks as necessary. If OFFSET is negative then it starts that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If LENGTH is negative, removes the elements from OFFSET onward except for -LENGTH elements at the end of the array. If both OFFSET and LENGTH are omitted, removes everything. If OFFSET is past the end of the array and a LENGTH was provided, Perl issues a warning, and splices at the end of the array.

The following equivalences hold (assuming  `$#a >= $i` )

|    |                                  |                                      |
|----|----------------------------------|--------------------------------------|
| 1. | <code>push(@a,\$x,\$y)</code>    | <code>splice(@a,@a,0,\$x,\$y)</code> |
| 2. | <code>pop(@a)</code>             | <code>splice(@a,-1)</code>           |
| 3. | <code>shift(@a)</code>           | <code>splice(@a,0,1)</code>          |
| 4. | <code>unshift(@a,\$x,\$y)</code> | <code>splice(@a,0,0,\$x,\$y)</code>  |
| 5. | <code>\$a[\$i] = \$y</code>      | <code>splice(@a,\$i,1,\$y)</code>    |