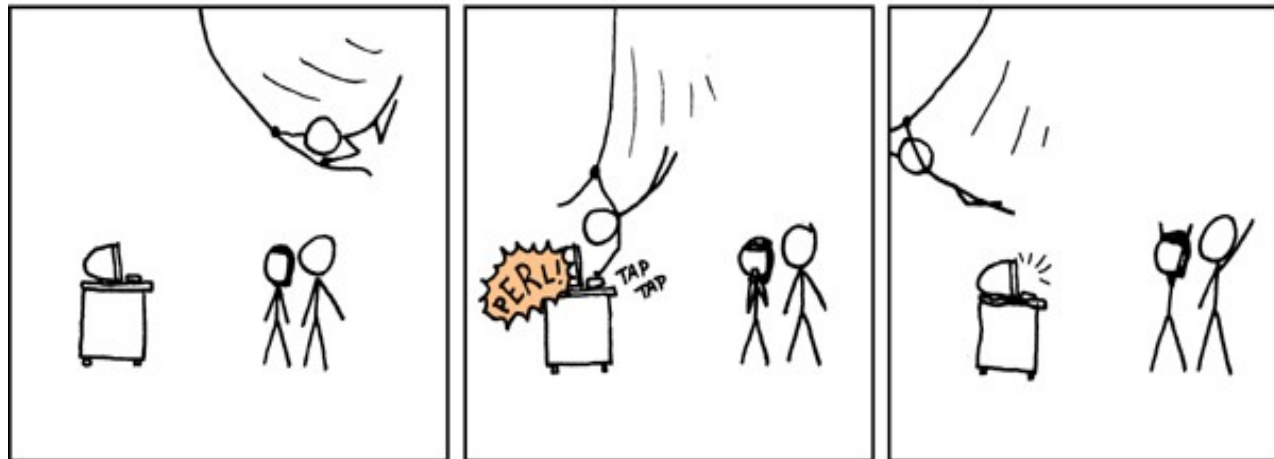


Programming Concepts: Strings

Paulo Penteado



Strings – definition

A **string** is a variable representing text, as a sequence (a string) of characters.

Every programming language has at least one standard variable type to represent and process strings.

It is one of the most often needed types, for everything. Exs:

- Inform the user
- File names
- Identifiers (elements, dates, names, programs, algorithms, objects, properties, etc.)
- File input and output (though not all data files are made with text)
- Building commands
- Most databases and web applications are string-centric

Among the basic variable types strings are the most complex to process.

Processing strings is not only *prints* and *reads*.

Strings - implementations

Programming languages vary widely in how they support strings.

There are **dynamic** and **static** strings:

- **Static** strings have a set number of characters (or maximum number of characters), which **cannot be changed**.
 - Trying to access characters beyond the end of the string can cause varied results: truncated strings, program dying (segfault), strings extending into other variables, or even big security holes.
 - Exs: C, Fortran
- **Dynamic strings**: the number of characters can be changed at any time, with no preset limits.
 - Exs: C++, Java, IDL, R, Python, **Perl**

Some languages have different types for individual characters and strings (made of zero or more characters): C, C++, Java.

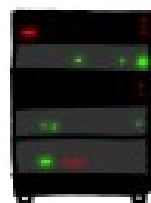
Some languages have several types to handle strings, with differing functionality.

HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE?
IF SO, REPLY "POTATO" (6 LETTERS).



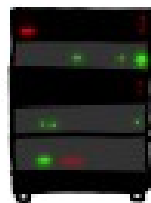
this page about boats. User Erica requires
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
Meggie (chrome user) sends this message: "H



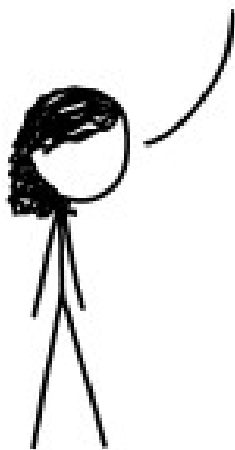
this page about boats. User Erica requires
secure connection using key "4538538374224"
User Meg wants these 6 letters: **POTATO**. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
Meggie (chrome user) sends this message: "H



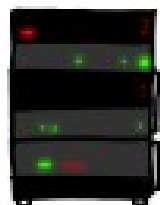
POTATO



SERVER, ARE YOU STILL THERE?
IF SO, REPLY "HAT" (500 LETTERS).



a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User



HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User

a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User



Strings - encoding

What makes up a string?

- Computers only “know” numbers (in binary).
- Nothing makes the contents of a variable or file intrinsically text. They are only 0s and 1s.
- The mapping between binary numbers and text **is determined by the encoding**, just like integer and real numbers are also encoded into binary digits.
- **Most languages assume a specific encoding**; some have different types for different encodings, and some may use string objects that can produce different encodings.

In ancient times (1980s) encoding was always the same: **ASCII** (*American Standard Code for Information Interchange*):

- 1 byte (7 or 8 bits) per character - 2^8 (256) or 2^7 (128) different characters.
- A standard table defines which character is encoded by each number in the range 0-127:

String encodings - ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

String encodings - ASCII

Not all ASCII character are visible (*printable*). Some are whitespace (space, tabs, etc.), other are some form o control character (null, CR, LF, etc.).

Zero is reserved for control, meaning either an empty string (made of only a zero), or, in some cases (C), the end of a string.

Characters 128-255 **are not in the ASCII standard**. The characters vary with the chosen ASCII extension.

ASCII is the simplest encoding in use:: characters always have the same size in memory (1 byte), and are easily read, processed and converted to/from numbers.

ASCII still is the most common encoding in scientific programming, but not the only one.

Line termination varies among systems. The most common choices:

- Unix-like systems (Linux, Max OS X): LF (**L**ine**F**eed; ASCII 10)
- Windows: CR (**C**arriage **R**eturn; ASCII 13) followed by LF (ASCII 10)
- Mac OS 9 and earlier: CR (ASCII 13)

ASCII does not mean the same as “text file”.

In recent years, **Unicode** encoding, in its many forms, is becoming more widespread.

String encodings - ASCII

Why not always use ASCII?

It is not enough. It does not contain, for instance:

- Modified characters (diacritical marks, cedilla)
- Math symbols (beyond the very basic $+ - \cdot * / \wedge ! \% > < =$)
→ Ex: $\mathbb{R} \mathbb{Z} \forall \partial \exists \Sigma \int \phi \pm \cong \geq \leq \times \infty \nabla \neq$
- Physical symbols
→ Ex: $\text{Å} \mu \odot \oplus$
- Greek letters
- Other symbols
→ Ex: $\rightarrow \leftrightarrow \rightleftharpoons \Rightarrow \text{€}^{\text{a}} \text{°} \text{£} \text{¥} \text{¿} \text{¡}$
- Characters from other languages (including those of many symbols, such as the forms used for Chinese and Japanese).

String encodings - Unicode

How to overcome the ASCII limitations?

The only widely used standard today is **Unicode**.

Developed to be “the one code”, with “every” character from “every” language, with metadata (**data describing the characters**).

It is not immutable, additions are decided by the Unicode Consortium (<http://www.unicode.org/>).

String encodings - Unicode

There are two parts to Unicode: the catalog (only one) and the encodings (many):

The catalog is independent from encodings:

“In Unicode, the letter A is a platonic ideal. It's just floating in heaven: A

*This platonic A is different than B, and different from a, but the same as **A** and A and A.*

The idea that A in a Times New Roman font is the same character as the A in a Helvetica font, but different from "a" in lower case, does not seem very controversial, but in some languages just figuring out what a letter is can cause controversy.

Is the German letter ß a real letter or just a fancy way of writing ss? If a letter's shape changes at the end of the word, is that a different letter? Hebrew says yes, Arabic says no.

Anyway, the smart people at the Unicode consortium have been figuring this out for the last decade or so, accompanied by a great deal of highly political debate, and you don't have to worry about it. They've figured it all out already.”

from The absolute minimum every software developer absolutely, positively, must know about Unicode and character sets (no excuses!),

<http://www.joelonsoftware.com/articles/Unicode.html>

String encodings - Unicode

The catalog has data about the characters, which are used in queries and to identify them, **including names and properties**: *printable*, numeric, alphanumeric, capital, blank, language, math, etc.

Exs:

Unicode Character 'LATIN CAPITAL LETTER A' (U+0041)

Name	LATIN CAPITAL LETTER A
Block	Basic Latin
Category	Letter, Uppercase [Lu]
Combine	0
BIDI	Left-to-Right [L]
Mirror	N
Index entries	Latin Uppercase Alphabet, Uppercase Alphabet, Latin Capital Letters, Latin
Lower case	U+0061
Version	Unicode 1.1.0 (June, 1993)



Unicode Character 'INTEGRAL' (U+222B)


Name	INTEGRAL
Block	Mathematical Operators
Category	Symbol, Math [Sm]
Combine	0
BIDI	Other Neutrals [ON]
Mirror	Y
Index entries	Integral Signs, INTEGRAL
See Also	latin small letter esh U+0283
Version	Unicode 1.1.0 (June, 1993)




(results from <http://www.fileformat.info/info/unicode/char/search.htm>)

String encodings - Unicode

Unicode Character 'ANGSTROM SIGN' (U+212B)

Name	ANGSTROM SIGN	
Block	Letterlike Symbols	
Category	Letter, Uppercase [Lu]	
Combine	0	
BIDI	Left-to-Right [L]	
Decomposition	LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5)	
Mirror	N	
Old name	ANGSTROM UNIT	
Index entries	ANGSTROM SIGN	
Lower case	U+00E5	
Comments	non SI length unit (=0.1 nm) named after A. J. Ångström, Swedish physicist, preferred representation is U+00C5	
Version	Unicode 1.1.0 (June, 1993)	

Unicode Character 'GREEK SMALL LETTER ZETA' (U+03B6)

Name	GREEK SMALL LETTER ZETA	
Block	Greek and Coptic	
Category	Letter, Lowercase [Ll]	
Combine	0	
BIDI	Left-to-Right [L]	
Mirror	N	
Upper case	U+0396	
Title case	U+0396	
Version	Unicode 1.1.0 (June, 1993)	

(results from <http://www.fileformat.info/info/unicode/char/search.htm>)

String encodings - Unicode

Unicode Character 'NABLA' (U+2207)

Name	NABLA
Block	Mathematical Operators
Category	Symbol, Math [Sm]
Combine	0
BIDI	Other Neutrals [ON]
Mirror	N
Index entries	difference, backward, backward difference, del, NABLA
Comments	backward difference, gradient, del, used for Laplacian operator (written with superscript 2)
See Also	white down-pointing triangle U+25BD
Version	Unicode 1.1.0 (June, 1993)



Java Data

string.toUpperCase()	∇
string.toLowerCase()	∇
Character.UnicodeBlock	MATHEMATICAL_OPERATORS
Character.charCount()	1
Character.getDirectionality()	DIRECTIONALITY_OTHER_NEUTRALS [13]
Character.getNumericValue()	-1
Character.getType()	25
Character.isDefined()	Yes
Character.isDigit()	No
Character.isIdentifierIgnorable()	No
Character.isLetter()	No
Character.isLetterOrDigit()	No
Character.isLowerCase()	No
Character.isWhitespace()	No
(...)	

(results from <http://www.fileformat.info/info/unicode/char/search.htm>)

Strings – Unicode encodings

There are two common encodings in the western world:

- **UTF-8**
- **ISO 8859-1** (also called *Latin 1*)

Unicode text, by itself, does not inform what is the encoding. The same sequence of bytes can mean different texts, depending on the encoding assumed.

- Software may **assume, ask or guess** the encoding.

All common Unicode encodings have ASCII as a subset: the 128 ASCII characters are coded identically to ASCII.

- **Unicode software writing only ASCII characters produces exactly the same output as ASCII.**

Strings – Unicode support

Languages vary widely

- **Do not know Unicode** (only use ASCII): C, Fortran
- **Use ASCII natively** (including for sourcecode), but have some variable types and libraries to process Unicode: C, C++, IDL, R.
- **Use Unicode natively** (including in sourcecode), and have extensive Unicode string support: Java, Python, Perl

Often (even when Unicode can be used in sourcecode), Unicode characters are written through ASCII with escape codes:

IDL> `p=plot(/test,title='!Z(00C5,222B)')` produces Åf

C, C++, Java, Python: `"\u2207"` produces ∇

References:

Characters vs. bytes

<http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF>

The absolute minimum every software developer absolutely, positively must know about Unicode and character sets (no excuses!)

<http://www.joelonsoftware.com/articles/Unicode.html>

Unicode character search

<http://www.fileformat.info/info/unicode/char/search.htm>

Strings – basic processing

Spend some time learning the toolkit your language provides.

Most common operations (in IDL syntax):

- **Concatenation**

```
IDL> a= 'some'
```

```
IDL> b=a+' string'
```

```
IDL> help,b
```

```
B                STRING      = 'some string'
```

- **Sorting**

```
IDL> help,a,b
```

```
A                STRING      = 'some'
```

```
B                STRING      = 'some string'
```

```
IDL> print,b gt a
```

```
1
```

```
IDL> c=[a,b,'9','Some',' some','some other string']
```

```
IDL> print,c[sort(c)],format='(A)'
```

```
some
```

```
9
```

```
Some
```

```
some
```

```
some other string
```

```
some string
```

Strings – basic processing

- Logical value:

Empty string (*null string*) is false, the rest is true:

```
IDL> c=''
```

```
IDL> if c then print,'c is not empty string' else print,"c is null  
string ('')"  
c is null string ('')
```

```
IDL> c='a'
```

```
IDL> if c then print,'c is not empty string' else print,"c is null  
string ('')"  
c is not empty string
```

Whitespace is not the same as empty string:

```
IDL> c=' '
```

```
IDL> if c then print,'c is not empty string' else print,"c is null  
string ('')"  
c is not empty string
```

Strings – basic processing

- **Substrings**

```
IDL> print, strmid('abcdefg', 3, 2)  
de
```

Some languages allow the use of indices to select substrings

- C, C++, Fortran, Python
- Ex: (Python)

```
>>> s="abcde"  
>>> print(s[2:5])  
cde
```

In IDL 8.4:

```
IDL> a='abcdefg'  
IDL> a.substring(2,5)  
cdef
```

- **Search for characters or substrings**

```
IDL> print, strpos('abcdefg', 'de')  
3
```

```
IDL> a='abcdefg'  
IDL> a.indexof('d') (IDL 8.4)  
3
```

Strings – basic processing

- Others

```
IDL> print, strlen('1234567')  
7
```

```
IDL> print, strlen(' 1234567 ')  
9
```

Measuring string length includes
whitespace.

```
IDL> help, strtrim(' 1234567 ', 2)  
<Expression>    STRING    = '1234567'
```

```
IDL> print, strupcase('abcdEF')  
ABCDEF
```

```
IDL> print, strjoin(['a', 'b', 'c'], '~')  
a~b~c
```

```
IDL> a='some random text' (IDL 8.4)  
IDL> a.replace('random', 'specific')  
some specific text
```

```
IDL> print, strsplit('temperature=19.8/K', '=/', /extract), format='(A)'  
temperature  
19.8  
K
```

Strings – creation from other types

Every time you see a number, it was converted to a string. Exs (DL):

```
IDL> print, [-1, 0, 9]
```

-1 0 9

```
IDL> print, 1d0, 1B, 1.0
```

1.0000000 1 1.000000

```
IDL> help, string(1d0, 1B, 1.0)
```

```
<Expression>    STRING    =    '1.0000000    1            1.000000'
```

```
IDL> printf, unit, dblarr(3, 4, 3) —————▶ Puts variables in a file, as strings
```

Strings



Strings – explicit formatting

Often, the default way a string is created from a variable is not adequate (number of digits, use of scientific notation, spacing, etc.)

In such cases, one must specify how to create the string (by a format).

Each language has its way to specify a format, but there are two common standards: C-like and Fortran-like.

Strings – explicit formatting

Fortran style

IDL:

```
IDL> print, 1d0+1d-9          No explicit format (default)
      1.00000000
```

```
IDL> print, 1d0+1d-9, format='(E16.10) '
1.00000000010E+00
```

```
IDL> print, 'x=', 1d0+1d-9, format='(A0, F16.13) '
X= 1.00000000010000
```

C (“printf”) style

IDL:

```
IDL> print, format='(%"x=%16.10e" )', 1d0+1d-9
x=1.00000000010e+00
```

Python:

```
In [20]: print("x=%16.10e" % (1e0+1e-9))
x=1.00000000010e+00
```

Strings - Fortran-style formatting

(just the main specifiers)

Ex (IDL):

```
IDL> print, 'x=', 1d0+1d-9, format='(A0, F16.13)'  
X= 1.00000000010000
```

Code	Meaning	Example(s)
A	String	'(A)', '(10A)'
I	Integer (decimal)	'(I)', '(10I)', '(-2I)'
B	Integer (binary)	'(B)', '(10B)'
Z	Integer (hexadecimal)	'(Z)', '(10Z)'
O	Integer (octal)	'(O)', '(10O)'
F	Real (fixed point)	'(F)', '(F5.2)'
E, D	Real (floating point)	'(E)', '(D16.10)'
G	Real (fixed or floating, depending on value)	'(G)', '(G10)'
""	String literal	'("x=", I10)'
X	blanks	'(A, 10X, I)'

There are modifiers for signs, exponents, leading zeros, line feed, etc.

Strings – C-style formatting (*printf*)

(just the main specifiers)

String with fields to be replaced by values, marked by codes with %

Ex. (Python):

```
In [20]: print("x=%16.10e" % (1e0+1e-9))  
x=1.0000000010e+00
```

Code	Meaning	Example(s)
d, i	Integer, decimal (<i>int</i>)	%d, %5d, %+05d
u	Integer, unsigned (<i>unsigned int</i>)	%u, %7u
f, F	Real, fixed-point (<i>double, float</i>)	%f, %13.6f
e, E	Real, floating point (<i>double, float</i>)	%e, %16.10e
g, G	Real, either fixed or floating point, depending on value (<i>double, float</i>)	%g, %7.3G
x, X	Integer, unsigned, hexadecimal (<i>unsigned int</i>)	%x, %10X
o	Integer, unsigned, octal (<i>unsigned int</i>)	%o, %5o
s	String (<i>string</i>)	%s, %10s
c	Character (<i>char</i>)	%c
p	Pointer – C-style - (<i>void *</i>)	%p
%	Literal %	%%

There are modifiers for signs, exponents, leading zeros, line feed, etc.

Strings – implicit conversion to other types

Exs (IDL):

```
IDL> help,fix(['17',' 17 ','17.1',' -17 ','9 8'])  
<Expression>      INT      = Array[5]
```

```
IDL> print,fix(['17',' 17 ','17.1',' -17 ','9 8'])  
      17      17      17      -17      9
```

```
IDL> print,double(['17',' 17 ','17.1',' -17 ','9 8'])  
      17.000000      17.000000      17.100000      -17.000000  
      9.00000000
```

```
IDL> readf,unit,a,b,c,d
```

```
IDL> a=0d0
```

```
IDL> b=0.0
```

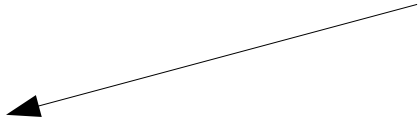
```
IDL> c=0
```

```
IDL> reads,'17.1d0 18.9d0 -9',a,b,c
```

```
IDL> help,a,b,c
```

```
A      DOUBLE      =      17.100000  
B      FLOAT       =      18.9000  
C      INT         =      -9
```

Converts the string into the types of the variables
a, b, c, d



Strings – conversion to other types

When default conversion is not enough, a format can be specified

```
IDL> a=0d0
```

```
IDL> b=0.0
```

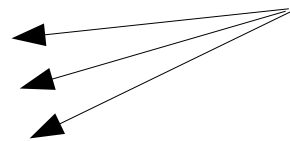
```
IDL> c=0
```

```
IDL> reads, '17.1d0 something 18.9d0, -9', a, b, c
```

```
% READS: Input conversion error. Unit: 0, File: <stdin>
```

```
% Error occurred at: $MAIN$
```

```
% Execution halted at: $MAIN$
```



Variables have to be created, to determine the types for the conversion

It did not work, because alone it does not know what to do with the “something”. Using a format:

```
IDL> reads, '17.1d0 something 18.9d0, -9',  
a, b, c, format='(D6.1, 11X, D6.1, 1X, I)'
```

```
IDL> help, a, b, c
```

A	DOUBLE	=	17.100000
B	FLOAT	=	18.9000
C	INT	=	-9

The format instructed IDL to read a double (**D6.1**), skip 11 characters (**11X**), read a double (**D6.1**), skip one character (**1X**), and read an integer (**I**).

Strings – other examples

- **Simple tests** (ex. IDL):

```
IDL> str=['a.fits', 'a.FITS', 'a.fitsa', 'ab.fits', 'abc.fits']
```

```
IDL> print, strmatch(str, '*.fits')  
  1   0   0   1   1
```

```
IDL> print, strmatch(str, '*.fits', /fold_case)  
  1   1   0   1   1
```

```
IDL> print, strmatch(str, '*.fits*', /fold_case)  
  1   1   1   1   1
```

```
IDL> print, strmatch(str, '?*.fits')  
  1   0   0   0   0
```

```
IDL> print, strmatch(str, '??*.fits')  
  0   0   0   1   0
```

Strings – other examples

- **Encodings** (Ex. Python 3):

```
>>> s="infinite money: ∞\N{euro sign}"
```

```
>>> print(s)
infinite money: ∞€
```

```
>>> print(ascii(s))
'infinite money: \u221e\u20ac'
```

```
>>> print(s.encode('utf-8'))
b'infinite money: \xe2\x88\x9e\xe2\x82\xac'
```

Regular expressions - definition

Regular expressions, (regex, regexp) are the most powerful tool to specify properties of strings.

Regex are a language, implemented similarly on most programming languages.

What are they for?

The interpreter (**regular expression engine**) gets the string and the expression, and determines whether the string *match* that expression.

In some cases, the interpreter can also inform which parts of the string match which part of the regex, and extract these parts.

Regular expressions – use cases

- **Separate parts of strings**

→ Find lines with names, values and comments, and extract these pieces:

Scalar with a comment (as in a FITS file):

```
'SLITPA = 351.979 / Slit position angle'
```

1D array spanning several lines

```
'BAND_BIN_CENTER = (0.350540, 0.358950, 0.366290, 0.373220, 0.379490,  
0.387900, 1.04598)'
```

Scalars in different formats:

```
'Total Mechanical Luminosity: 1.5310E+03'  
'resources_used.walltime=00:56:03'
```

Pieces of names:

```
'60.63 1.7836E-20 2.456 T FeIX((3Pe)3d(2PE)4p_1Po-3s2_3p6_1Se)'
```

Dates, separating year, month, day, hour, minute, second:

```
'DATE-OBS= '2006-12-18' / universal date of observation'  
'DATE_TIME = 2010-07-19T16:10:32'  
'START_TIME = "2006-182T22:51:02.850Z"'
```

Regular expressions – use cases

- **Separate pieces of strings**

- Extract pieces of files names, because they mean something about the file contents:

```
'spec/dec18s0041.fits'  
'scam/dec18i0054.fits'  
'15_7_mts_hm/pixelh_mr15.sav'  
'15_7_mts_hw/pixelh_mr15.sav'  
'16_3_mts_hw/pixelb_mr16.sav'  
'readmodel5l_-1_0.00010000_1.0000_r05_030_08196_0.100000_0.05000000_10.00.eps'
```

- **Determine whether a string represents a number** (integer or real, fixed or floating point).

- **Locate identifiers in file contents. Exs:**

- Catalog identifiers in the middle of the text
- Web addresses (http, ftp, etc.)
- File names
- Form values
- Data elements in text

Regular expressions – simple example

Ex. (IDL): find file names with some property:

```
IDL> print,files,format='(A)'  
CM_1477475933_1_vis.cub  
CM_1477476864_1_ir.cub  
CM_1477476864_1_irg.cub  
CM_1477476864_1_vis.cub  
CM_1477477826_1_ir.cub  
CM_1477477826_1_irg.cub  
CM_1477477826_1_vis.cub  
mosaic2.cub
```

```
IDL> print,stregex(files, '.+irg\.cub',/boolean)  
0 0 1 0 0 1 0 0
```

The regex `'.+irg\.cub'` specifies:

- One or more occurrences (+) of any character (.),
- Followed by an occurrence of `irg.cub` (the period is escaped (with the backslash), to be understood as a literal period, not its special meaning).

This use of the interpreter (`stregex`) returns a true/false result for each string it gets (`files`), telling whether that string matches the regex.

This use of regex is overkill. It would have been easier to do `strmatch(files, '*irg.cub')`.

Regular expressions – simple example

Ex. (IDL): Determine which strings represent a date in the format **yyyy-mm-dd**:

```
IDL> strs=['20100201', '2010-02-01', '2010-2-1', 'aaaa-mm-dd', 'T2010-02-01J']
IDL> print, stregex(strs, '[0-9]{4}-[0-9]{2}-[0-9]{2}', /boolean)
      0      1      0      0      1
```

This regex means:

- 4 repetitions (**{4}**) of digits (characters in the range **[0-9]**),
- Followed by (-),
- Followed by 2 repetitions (**{2}**) of digits (**[0-9]**),
- Followed by (-),
- Followed by 2 repetitions (**{2}**) of digits (**[0-9]**).

A slightly more complex regex could match the 3 date formats above. It could also reject the last expression (which has extra characters before and after the date).

Regular expressions - rules

A regex with “normal”* characters specifies a string with those characters, in that order.

- Ex: 'J' is a regex that matches any string containing J. 'JA' is a regex that only matches strings containing 'JA'.
- Exs. (IDL):

```
IDL> strs=['J', 'JJJJJ', 'aJA', 'j', 'aJa']
```

```
IDL> print, stregex(strs, 'J', /boolean)
  1   1   1   0   1
```

```
IDL> print, stregex(strs, 'JA', /boolean)
  0   0   1   0   0
```

*some characters have special meaning in regular expressions (shown ahead).

Regular expressions – special characters

These symbols have special meanings. To represent literally that symbol, it must be escaped with a `\`:

Symbol	Meaning	example	Match
<code>\</code>	Escape: the following character must be interpreted literally, not by its special meaning.	'\?'	'?', 'a?a'
<code>.</code>	Any character	'a.b'	'ajb', 'aab', 'abb', 'jafbc'
<code>+</code>	One or more repetitions of the preceding element.	'a+b'	'ab', 'aab', 'bab', 'baabh'
<code>()</code>	Subexpression: groups characters so that several of them are affected by the modifiers (like parenthesis in math).	'(ab)+c'	'abc', 'ababc', 'dabababcg'
<code>*</code>	Zero or more repetitions of the preceding element.	'a*b'	'ab', 'b', 'aab', 'caaabg'
<code>?</code>	Zero or one occurrence of the preceding element	'a?b'	'b', 'ab', 'cabd', 'cbd'
<code> </code>	Alternation: either one of the two elements.	'a bc'	'ac', 'bc', 'jacd', 'jbcd'
<code>{n}</code>	Exactly n repetitions of the preceding element.	'a{2}b'	'aab', 'daaabg'
<code>{n1, n2}</code>	From n1 to n2 repetitions of the preceding element.	'a{1, 2}b'	'ab', 'aab', 'aaab', 'gaaabbd'
<code>^</code>	Anchor: beginning of string.	'^ab'	'ab', 'abb'
<code>\$</code>	Anchor: end of string.	'ab\$'	'ab', 'aab'
<code>[]</code>	Value set (shown ahead)		

Regular expressions – value sets

[] means a set a value, which may be:

- **A set of things to match.**

- Ex: '[abc]' means any of the characters **a,b,c**: Ex. Matches: 'a', 'b', 'c', 'ab', 'ha'.

- **A set of things not to match**

- '[^abc]' means anything other than **a, b** ou **c**: Ex matches: 'd', 'jgs', 'gg'.

- **Value ranges**

- '[0-9]' any digit
- '[0-9a-zA-Z]' any digit or letter

- **Value classes**

- Special names for some types of values (in IDL, these come delimited by [::]):
- ex: '[:digit:]' means the same as '[0-9]'.

Regular expressions – value classes

Class	meaning
<code>alnum</code>	Alphanumeric characters: 0-9a-zA-Z
<code>alpha</code>	Alphabetic chracters: a-zA-Z
<code>cntrl</code>	ASCII control characters (not printable, codes 1 to 31 and 127).
<code>digit</code>	Digits (decimal): 0-9
<code>graph</code>	Printable characters: ASCII 33 to 126 (excl. space).
<code>lower</code>	Lower case letters: a-z
<code>print</code>	Printable characters “imprimíveis” (visible plus space): ASCII 32 to 126 .
<code>punct</code>	Punctuation: !"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~
<code>space</code>	Whitespace: space, tab, vertical tab, CR, LF (ASCII 32 and 9-13).
<code>upper</code>	Capital letters: A-Z
<code>xdigit</code>	Hexadecimal digits: 0-9A-Fa-f
<code><</code>	Beginning of the word (“word” meaning a sequence of non-space characters).
<code>></code>	End of word.

These are just the main classes. Some languages have many others.

Regular expressions - examples

Determine whether a string represent a number. (Ex. IDL):

```
IDL> str=['9', '-18', ' 8.75', '-8.1', '.2', '-.459', '1.3E9', '-9.8d7', 'a18.8d0', '3.2f5']
```

•Integers:

```
IDL> intexpr='^[-+]?[0-9]+$'
```

Optional
sign

1 or
more
digits

```
IDL> print, stregex(str, intexpr, /boolean)
```

```
1 1 0 0 0 0 0 0 0 0
```

•Floating point: Fixed-point number or floating-point number (mantissa and exponent)

```
IDL> fpexpr='^[-+]?([0-9]*\.[0-9]+|([0-9]+\.[0-9]*))([eEdD][-+]?[0-9]+)?$'
```

Optional
sign

0 or more digits,
optionally followed by
a period, plus 1 or
more digits

or

1 or more digits,
optionally followed by
a period, plus 0 or
more digits

Optional exponent:
letter (e/d), followed by
optional sign, followed
by 1 or more digits

```
IDL> print, stregex(str, fpexpr, /boolean)
```

```
1 1 0 1 1 1 1 1 0 0
```

Regular expressions - extraction

Regular expressions can also be used **to extract pieces of the string, that matched pieces of the expression.**

Ex (IDL): Determine whether a string contains a date, in any of these formats

```
IDL> dates=['2011-01-31', '2011 1 31', '2011/01/31', 'something done on  
y2011m1d31 with something']
```

And extract the dates from the strings

```
IDL> expr='[0-9]{4}.[0-9]{1,2}.[0-9]{1,2}'
```

(4 digits)(any separator)(1 to 2 digits)(any separator)(1 to 2 digits)

```
IDL> print, stregex(dates, expr, /extract), format='(A)'
```

```
2011-01-31
```

```
2011 1 31
```

```
2011/01/31
```

```
2011m1d31
```

Now, how do we extract each piece (year, month, day)? One operation for each part?

- Could be, much a regex does it all.

Regular expressions - extraction

Ex (IDL): :

- In this case, to make for a smaller regex, we assume a simple format: (**yyyy-mm-ddThh:mm:ss.fff**).

```
IDL> str='Stuff observed on 2011-01-31T12:39:24.983 with some instrument'
IDL> expr='([0-9]{4})-([0-9]{2})-([0-9]{2})T([0-9]{2}):([0-9]{2}):([0-9]{2}\.[0-9]{3})'
```

(4 digits) - (2 digits) - (2 digits) T (2 digits) : (2 digits) : (2 digits).3 digits

```
IDL> pieces=stregex(str,expr,/extract,/subexpr)
```

```
IDL> print,pieces,format='(A)'
```

2011-01-31T12:39:24.983	Whole match
2011	First subexpr
01	Second subexpr
31	Third subexpr
12	Fourth subexpr
39	Fifth subexpr
24.983	Sixth subexpr

```
IDL>
```

```
d=julday(pieces[2],pieces[3],pieces[1],pieces[4],pieces[5],pieces[6])
```

```
IDL> print,d,format='(F16.6)'
```

```
2455593.027372
```

Regular expressions – use case

Automate filling out a web form and reading the results

Object Observability

See also: [Object Observability](#) - [Airmasses](#) - [Daily Almanac](#) - [SK](#)

This tool provides object observability tables based on site, object time, including daylight saving times when applicable.

Select site, object coordinates and observing period; then press

More detailed information is provided in a separate document



The ESO Sky Calendar Tool

[HOME](#) [INDEX](#) [SEARCH](#) [HELP](#) [NEWS](#)

Site:

Dates (yyyy mm dd):

From: To:

Object Coordinates (J2000)

RA: Dec:

[compute](#)

See also [Object Observability](#) - [Airmasses](#) - [Daily Almanac](#) - [Ephemerides](#)

Observability for 17:42:15 00 00 -20:45:13 00 00

Paranal Observatory (VLT)

RA & dec: 17 42 15.0, -20 45 13, epoch 2000.0
Site long&lat: +4 41 36.8 (h.m.s) West, -24 37 30 North.

Shown: local eve. date, moon phase, hr ang and sec.z at (1) eve. twilight, (2) natural center of night, and (3) morning twilight; then comes number of nighttime hours during which object is at sec.z less than 3, 2, and 1.5. Night (and twilight) is defined by sun altitude < -18.0 degrees.

Date (eve)	moon	eve		cent		morn		night hrs@sec.z:		
		HA	sec.z	HA	sec.z	HA	sec.z	<3	<2	<1.5
2015 Jun 1	F	-6 23	16.2	-1 04	1.0	+4 15	1.9	9.4	8.6	7.0
2015 Jun 15	N	-5 27	3.7	-0 06	1.0	+5 15	3.2	10.3	8.7	7.0
2015 Jul 1	F	-4 20	2.0	+1 00	1.0	+6 21	14.4	9.5	8.7	7.0
2015 Jul 15	N	-3 21	1.4	+1 58	1.1	+7 16	down	8.5	7.7	6.8
2015 Jul 30	F	-2 16	1.2	+2 57	1.3	+8 10	down	7.4	6.6	5.7
2015 Aug 13	N	-1 16	1.1	+3 51	1.7	+8 58	down	6.4	5.6	4.7
2015 Aug 29	F	-0 08	1.0	+4 50	2.5	+9 48	down	5.3	4.5	3.6
2015 Sep 12	N	+0 52	1.0	+5 40	4.5	+10 29	down	4.3	3.5	2.6
2015 Sep 27	F	+1 57	1.1	+6 34	47.1	+11 12	down	3.2	2.4	1.5
2015 Oct 12	N	+3 03	1.4	+7 29	down	+11 54	down	2.1	1.3	0.4
2015 Oct 26	F	+4 08	1.8	+8 22	down	-11 25	down	1.0	0.2	0.0
2015 Nov 11	N	+5 25	3.6	+9 25	down	-10 35	down	0.0	0.0	0.0
2015 Nov 25	F	+6 33	37.0	+10 23	down	-9 47	down	0.0	0.0	0.0

Regular expressions – use case

HTML result (part of)

```
<h2>Observability for 17:42:15 00 00 -20:45:13 00 00</h2><b>Paranal Observatory (VLT)</b><p><pre>
```

```
RA & dec: 17 42 15.0, -20 45 13, epoch 2000.0
```

```
Site long&lat: +4 41 36.8 (h.m.s) West, -24 37 30 North.
```

```
Shown: local eve. date, moon phase, hr ang and sec.z at (1) eve. twilight, (2) natural center of night, and (3) morning twilight; then comes number of nighttime hours during which object is at sec.z less than 3, 2, and 1.5.
```

```
Night (and twilight) is defined by sun altitude < -18.0 degrees.
```

```
  Date (eve) moon      eve          cent          morn          night hrs@sec.z:
                HA  sec.z      HA  sec.z      HA  sec.z      <3    <2    <1.5
2015 Jun 1      F   -6 23  16.2   -1 04   1.0    +4 15   1.9    9.4    8.6    7.0
2015 Jun 15     N   -5 27   3.7    -0 06   1.0    +5 15   3.2   10.3    8.7    7.0
2015 Jul 1      F   -4 20   2.0    +1 00   1.0    +6 21  14.4    9.5    8.7    7.0
2015 Jul 15     N   -3 21   1.4    +1 58   1.1    +7 16  down    8.5    7.7    6.8
2015 Jul 30     F   -2 16   1.2    +2 57   1.3    +8 10  down    7.4    6.6    5.7
2015 Aug 13     N   -1 16   1.1    +3 51   1.7    +8 58  down    6.4    5.6    4.7
2015 Aug 29     F   -0 08   1.0    +4 50   2.5    +9 48  down    5.3    4.5    3.6
2015 Sep 12     N   +0 52   1.0    +5 40   4.5   +10 29  down    4.3    3.5    2.6
2015 Sep 27     F   +1 57   1.1    +6 34  47.1   +11 12  down    3.2    2.4    1.5
2015 Oct 12     N   +3 03   1.4    +7 29  down   +11 54  down    2.1    1.3    0.4
2015 Oct 26     F   +4 08   1.8    +8 22  down   -11 25  down    1.0    0.2    0.0
2015 Nov 11     N   +5 25   3.6    +9 25  down   -10 35  down    0.0    0.0    0.0
2015 Nov 25     F   +6 33  37.0   +10 23  down    -9 47  down    0.0    0.0    0.0
2015 Dec 10     N   +7 45  down   +11 28  down    -8 48  down    0.0    0.0    0.0
```

```
</pre>
```

Regular expressions – use case

Automate filling out a web form and reading the results

Exposure time calculation **HAD WARNINGS.**

WARNING MESSAGE: "Electrons per pixel due to background (1.8) is less than the recommended threshold of 20 electrons to avoid poor charge transfer efficiency (CTE). We suggest you consider CTE mitigation strategies described in the STIS Instrument Handbook."

[View results in tabular form](#)

Plots

Total Counts

Signal-to-noise

Input Target Spectrum

Throughput

Observed Target Spectrum

Detailed Information	Count rate (e-/s)	Total counts (e-)	Associated noise (e-)
Counts (box 7 pixels high)	(1 pixel)	(2.0 pix resel)	
Source	0.855	358.71	18.94
Background	0.119	49.96	7.07
Sky	8.641e-05	0.04	0.19
Dark Current	0.119	49.93	7.07
Read Out			29.63
Total in selected region	0.974	408.68	35.87
Brightest Pixel (single exposure) (at 4521.08 Å)	0.359	37.69	
Global Count Rates			
Source only	924.627	193,969.495	

Other tools – sed

Not everything requires writing a program.

Here are some useful command-line tools.

- They read and write the files continuously: **they never have the whole file in memory**
- Useful for large files

- **sed (Stream EDitor)**
 - Simple search and replace: Replace **one** with **two**
`sed -e 's/one/two/g' input_file.txt > output_file.txt`

 - Search and replace with regular expressions: change date from **mm-dd-yyyy** to **yyyy-mm-dd**:
`sed -e 's/\([0-9]*\) - \([0-9]*\) - \([0-9]*\) / \3 - \1 - \2/g'`
`input_file.txt > output_file.txt`

Other tools – awk

- **AWK** (Aho, Weinberger, Kernighan)

→ **Read a CSV file**

```
name, RA, Dec, mag  
obj1, 17.42589765, -30.5928234, 22.1  
obj2, 18.38947923, 0.292843424, 12.1  
obj3, 8.2389240, -50.22099923, 18.9
```

→ and select only its 2nd through 4th columns, where the last is <20:

```
awk -F, '{if ($4<20) {print $2", "$3", "$4}}' input.csv
```

```
18.38947923, 0.292843424, 12.1  
8.2389240, -50.22099923, 18.9
```

→ The same, but keeping the header line:

```
awk -F, '{if ($4<20 || NR==1) {print $2", "$3", "$4}}' input.csv
```

```
RA, Dec, mag  
18.38947923, 0.292843424, 12.1  
8.2389240, -50.22099923, 18.9
```


Some references

Software Carpentry Videos on Regular Expressions:

http://software-carpentry.org/4_0/regexp/

An Awk Primer/Awk Command-Line Examples

http://en.wikibooks.org/wiki/An_Awk_Primer/Awk_Command-Line_Examples

15 Practical Grep Command Examples In Linux / UNIX

<http://www.thegeekstuff.com/2009/03/15-practical-unix-grep-command-examples/>

Sed - An Introduction and Tutorial by Bruce Barnett

<http://www.grymoire.com/Unix/Sed.html>

This presentation is at

<http://www.ppenteado.net/pc>

