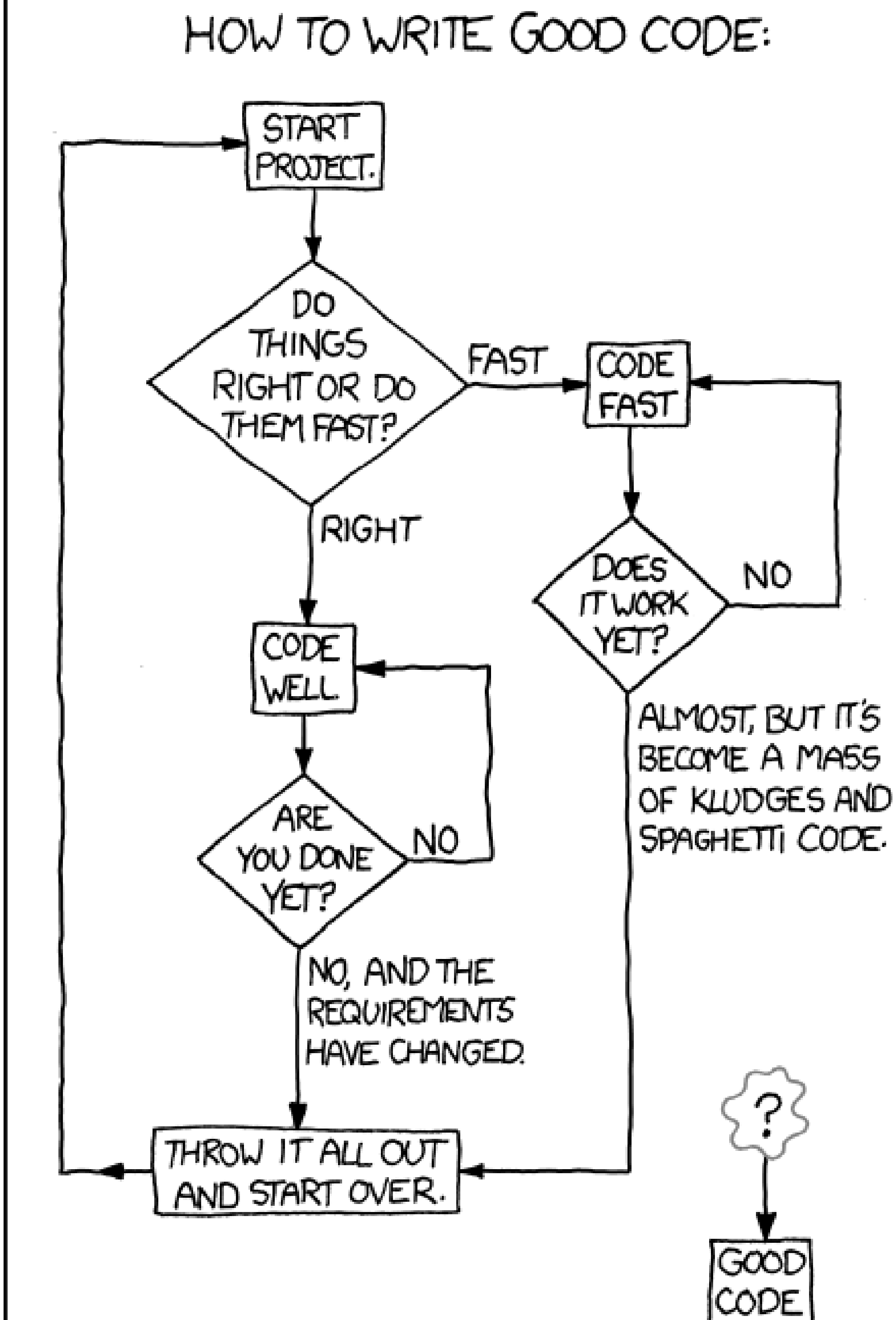
Programming Concepts: Variables

Paulo Penteado

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



(http://www.xkcd.org/844)

Variable types

What is a variable?

In the old days, just a name for a position in memory:

Instead of saying

> *Store integer **2** on position **47** (of the memory)*
> *Add integer **1** to the contents of position **47***
> *Print the contents of position **47***

One could say (in pseudocode*)

```
int number_of_days
number_of_days=2
number_of_days=number_of_days+1
print number_of_days
```

Much better to use *number_of_days* than memory positions:

- The name gives a cue to the meaning of the number
- More readable and portable

*No specific language

Types

A variable is much more elaborate than just a position in memory

If I put π in memory position 435, what is in there?

Types

A variable is much more elaborate than just a position in memory

If I put π in memory position 435, what is in there?

?

Types

A variable is much more elaborate than just a position in memory

If I put π in memory position 435, what is in there?

?

Position 435

No. More like:

...1001110010010010100**0**10000000010010010000111111011011000101001...

Types

A variable is much more elaborate than just a position in memory

- A variable is a representation of a **type** in one's program.

- A **type** is an abstract concept
  - Exs: integers, reals, strings (text), complex numbers, etc.

- **Internally, these concepts do not exist**: There are no reals in memory. There are (binary) representations of them, through rules defined by the type **real**.

- **This concept includes rules about their use**. Exs:

  - Adding 2 integers is not the same as adding 2 reals. The processor uses different algorithms.

  - **3/2** is **1**, while **3.0/2.0** é **1.5**

  - **acos(2.0)** does not exist for reals, but it does for complex numbers.

  - Strings can be coded in many different ways.

  - Rules for ordering string may differ (does capitalization matter? where do numbers go in the order?)

Types

A variable can be much more complicated than a number or a string

- **Container**: stores several values, by orders, name or hierarchy
  - Exs: vector, matrix, array, list, map, dictionary, tree, etc.
- **Several values of different types, organized** - Structure
- **Reference to another variable** - pointer
- **A representation of any complicated concept - object**
  - Data, resources and ways to operate on them
  - It is an active variable (“smart”): not just a static data store, it can do operations.

Common types

Some commonly used types (*standard, built-in, primitive*):

Python: int, long, float, complex, str

IDL: int, string, float, double, byte, complex, ptr, obj

Fortran: integer, character, logical, real, double precision, complex, pointer

C, C++, Java: int, char, float, double

SQL: int, small int, bool, float, double

These types are common in all languages, but are not necessarily the same. Exs:

- Python' s ***Float*** usually corresponds to a ***double*** in other languages (double precision)
- IDL's ***int*** has 16 bits, Fortran's ***integer*** usually has 32 bits
- Fortran's ***real*** might have 32 bits or 64 bits

Common types – differences for similar ideas

Types are not different only in the "nature" of the idea. Exs:

- Differently sized integers
  - **Byte**, **int8** (Numpy): 8 bits, stores integers from **0** to **255** (2^8-1)
  - **Short**, **int16** (Numpy): 16 bits, stores integers from -**32768** (-(2^{15})) to **32767** (2^{15}-1)
- Precision for reals: **single** and **double**:
  - **`1.0+1e-8 is 1.0`** (32bits, 6 or 7 significant digits)
  - **`1d0+1d-8 is 1.00000001`** (64 bits, ~14 significant digits)

The same function/operator is usually different with different types (ex. in IDL/Python):

- **`3/2`** is **`1`**, while **`3.0/2.0`** is **`1.5`**
- **`sqrt(-1.0)`** `is` **`-NaN`**, while **`sqrt(complex(-1.0))`** is **`(0.0,1.0)`**

Types – empty type

Just as zero did not exist until modern number systems, modern languages do have empty types, with important uses:

- **To indicate something is missing**
  - Ex: A list where each element tells which observations were taken of the corresponding target. Some targets may have no observations.

- **To indicate no results**
  - Ex: functions that query some data source, to indicate that nothing was found.

- **Undefined variables / elements**
  - Indicates an input argument must be replaced by defaults
  - Indicates some output argument is not required

Examples:

- **None** (Python)
- **!null** (IDL ≥8)
- **NULL** (R, C++, Perl)
- **null** (Java)

Types – empty type

Just as zero did not exist until modern number systems, modern languages do have empty types, with important uses:

- **To indicate something is missing**
  - Ex: A list where each element tells which observations were taken of the corresponding target. Some targets may have no observations.
- **To indicate no results**
  - Ex: functions that query some data source, to indicate that nothing was found.
- **Undefined variables / elements**
  - Indicates an input argument must be replaced by defaults
  - Indicates some output argument is not required

Example (Python):

```
In [55]: observations={'Vesta':10,'Pluto':7}

In [56]: print observations.get('Pluto')
7

In [57]: result=observations.get('Eris')

In [58]: if result is None:
   ....:     print 'There is no information about this object'
   ....:
?
```

Types – empty type

Just as zero did not exist until modern number systems, modern languages do have empty types, with important uses:

- **To indicate something is missing**
  - Ex: A list where each element tells which observations were taken of the corresponding target. Some targets may have no observations.
- **To indicate no results**
  - Ex: functions that query some data source, to indicate that nothing was found.
- **Undefined variables / elements**
  - Indicates an input argument must be replaced by defaults
  - Indicates some output argument is not required

Example (Python):

```
In [55]: observations={'Vesta':10,'Pluto':7}

In [56]: print observations.get('Pluto')
7

In [57]: result=observations.get('Eris')

In [58]: if result is None:
   ....:     print 'There is no information about this object'
   ....:
There is no information about this object
```

Number representations and their consequences

Numbers in variables are not the same as the mathematical concept.

A variable has limited memory. Therefore, a number's digits are limited:

- The amount of different numbers that can be stored is finite.
- The numbers that are representable are predefined by the type being used.
- The precision and range numbers can have are limited.

Basic number types (integers, reals) are usually the same as the native processor number types.

They usually have fixed memory size. Exs: 8, 16, 32, 64 bits (1, 2, 4, 8 bytes).

Each bit (*binary digit*) is a memory position, which can only hold either **0** or **1**.

A type with **n** bits can only hold 2^n different values.

The most common types have **256**, **65536**, **~**4.3×10^9 (4 giga, in binary sense), or **~**1.8×10^{19} (16 exa, in binary sense) different values.

Number representation - integers

There are types for positive (*unsigned*) and types for negative/positive.

Positive integers are simply the number in binary.

Ex: with 8 bits, there is room for only 0 to 255:

```
Decimal    memory representation
0          00000000
1          00000001
2          00000010
255        11111111
```

Types that can take negatives are (usually) the same, with ~half the numbers being positive, **at the beginning**, then the negatives:

Ex: with 8 bits, there is only room for -128 to +127:

```
Decimal    memory representation
0          00000000
1          00000001
127        01111111
-128       10000000
-127       10000001
-126       10000010
-2         11111110
-1         11111111
```

Integer representations – common names and sizes*

8 bits:

• byte	(IDL, only positives)
• byte	(Java)
• char	(C, C++)
• tinyint	(MySQL)

16 bits:

• int	(IDL)
• short int	(C++)
• short	(Java)
• smallint	(MySQL)

32 bits:

• integer	(Fortran, R)
• long	(IDL)
• int	(C, C++, Java, Python,MySQL)
• long int	(C, C++)

64 bits:

• long	(Python)
• long64	(IDL)
• bigint	(MySQL)

*In some languages, the standard does not specify which type is which size; each implementation may make different choices. The values above are the most common.

Integer representations – common names and sizes*

Literals* usually have a default type, and can be changed with modifiers (exs. IDL):

- 9 9 of the default integer type
- 8L 8 of type *long* (32 bits)
- 25B 25 of type *byte* (8 bits)
- 12UL 12 of type *unsigned long* (64 bits)

*constants that appear *literally* inside the code

Number representation – consequences (integers)

What happens if you try to put in a variable a number that does not fit in it?

- In a *byte* type, which only holds **0** to **255**, how much is **255B+1B**? What about **0B-1B**?
- In a *short* type, which only holds **-32768** to **+32767**, how much is **-32767S-2S**?

Ex. (Python):

```
In [1]: import numpy

In [2]: a=numpy.array((0,255),dtype='uint8')

In [3]: print a
?

In [4]: a[0]=a[0]-1

In [5]: a[1]=a[1]+1

In [6]: print a
?
```

Number representation – consequences (integers)

What happens if you try to put in a variable a number that does not fit in it?

- In a *byte* type, which only holds **0** to **255**, how much is **255B+1B**? What about **0B-1B**?
- In a *short* type, which only holds **-32768** to **+32767**, how much is **-32767S-2S**?

Ex. (Python):

```
In [1]: import numpy

In [2]: a=numpy.array((0,255),dtype='uint8')

In [3]: print a
[  0 255]

In [4]: a[0]=a[0]-1

In [5]: a[1]=a[1]+1

In [6]: print a
[255   0]
```

Number representation – consequences (integers)

What happens if you try to put in a variable a number that does not fit in it?

- In a *byte* type, which only holds **0** to **255**, how much is **255B+1B**? What about **0B-1B**?
- In a *short* type, which only holds **-32768** to **+32767**, how much is **-32767S-2S**?

There is an *overflow* (ou *rollover*). Like a car's odometer (ex. IDL):

```
IDL> a=255B
IDL> help,a
A               BYTE      =   255

IDL> a=a+1B
IDL> help,a
A               BYTE      =     0

IDL> print,0B-1B
 255

IDL> print,-32768S-1S
   32767
```

Internally (binary):

```
    11111111

    11111111
+   00000001
   100000000
=   00000000

   1000000000000000
-  0000000000000001
=  0111111111111111
```

Number representation – consequences (integers)

Not considering integer size is a common error:

Ex: In IDL, where default integers are type int (16 bits):

```
IDL> print,10^4
?

IDL> print,10^5
?
```

Ex: In Python:

```
In [28]: import numpy

In [29]: b=numpy.array((10,10),dtype='int16')

In [30]: print b
[10 10]

In [31]: b[0]=b[0]**4

In [32]: b[1]=b[0]*10

In [33]: print b
?
```

Number representation – consequences (integers)

Not considering integer size is a common error:

Ex: In IDL, where default integers are type int (16 bits):

```
IDL> print,10^4
   10000

IDL> print,10^5
  -31072
```

Ex: In Python:

```
In [28]: import numpy

In [29]: b=numpy.array((10,10),dtype='int16')

In [30]: print b
[10 10]

In [31]: b[0]=b[0]**4

In [32]: b[1]=b[0]*10

In [33]: print b
[ 10000 -31072]
```

Number representation – consequences (integers)

Not considering integer size is a common error:

Ex: In IDL, where default integers are type int (16 bits):

```
IDL> print,10^4
   10000

IDL> print,10^5
  -31072
```

The result for `10`^{5} is not wrong:

- **10^5** is larger than the largest integer that can fit in 16 bits (**32767**)
- After **32767** comes **-32768**, then **-32767**, etc.

With a larger type, there is no overflow for this number:

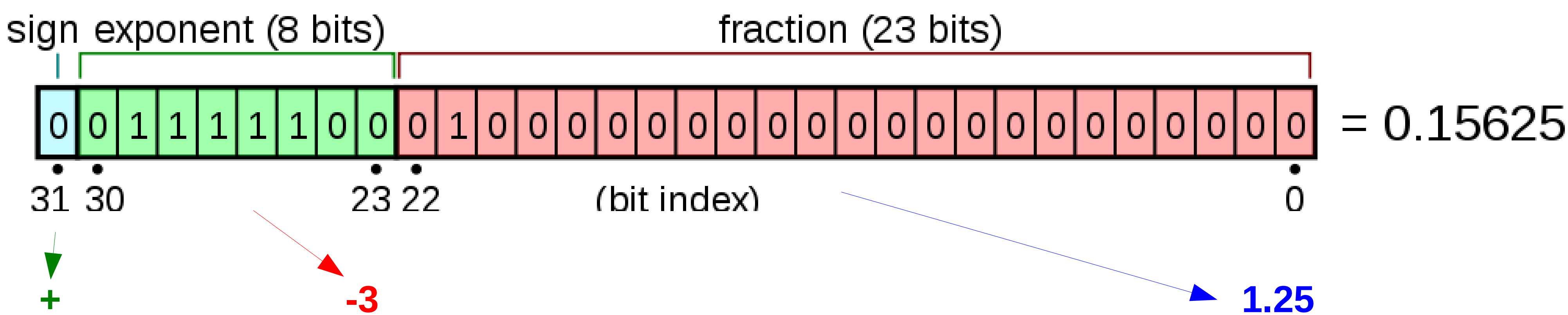
```
IDL> print,10L^4
   100000
```

Number representation - reals

Usual types come from the IEEE 754 standard for floating point number representation and manipulation:

- ***single precision / float / real*** (32 bits)
- ***double precision / double*** (64 bits)

Numbers are represented by a fraction signficand, and exponent and a sign, similarly to scientific notation (ex: **0.31416E1**), but with binary digits:



+ 1.25 × 2^{-3} = 0.15625

	Sign	Exponent	Fraction	Range	Significant digits (decimal)
Float	1 bit	8 bits	23 bits	~$\pm 10^{38}$	6-9
Double	1 bit	11 bits	52 bits	~$\pm 10^{308}$	15-17
Quad*	1 bit	15 bits	112 bits	~$\pm 10^{4932}$	33-36

*Rarely implemented

Number representation - reals

***Single precision floats* are common, but insufficient for scientific computing.**

Literals / strings such as 1.0 and 1e5 might be interpreted as *floats*.

***Doubles* might be written as 1.0d0 e 1d5. But a “d” in a string usually does not change its interpretation.**

Attention to the type used in literals: (Exs. Python):

```
In [65]: print 1/3
?

In [66]: print 1.0/3.0
?

In [67]: print 16**(1/2)
?

In [68]: print 16**(1.0/2.0)
?
```

Number representation - reals

Single precision floats **are common, but insufficient for scientific computing.**

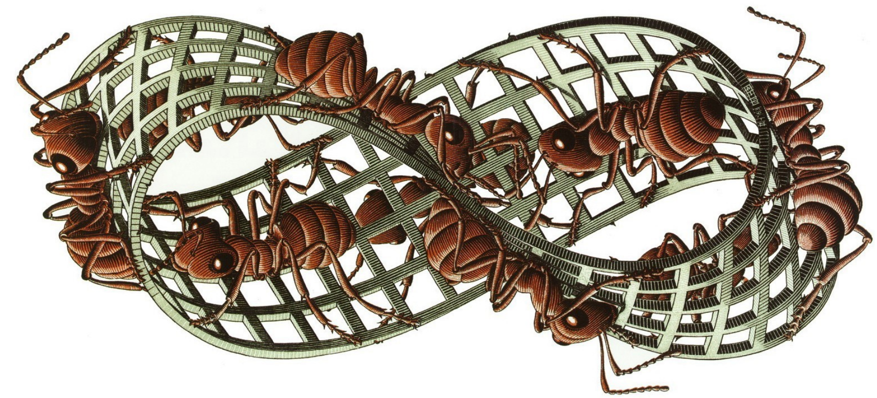
Literals / strings such as 1.0 and 1e5 might be interpreted as *floats*.

Doubles **might be written as 1.0d0 e 1d5. But a “d” in a string usually does not change its interpretation.**

Attention to the type used in literals: (Exs. Python):

```
In [65]: print 1/3
0

In [66]: print 1.0/3.0
0.333333333333

In [67]: print 16**(1/2)
1

In [68]: print 16**(1.0/2.0)
4.0
```

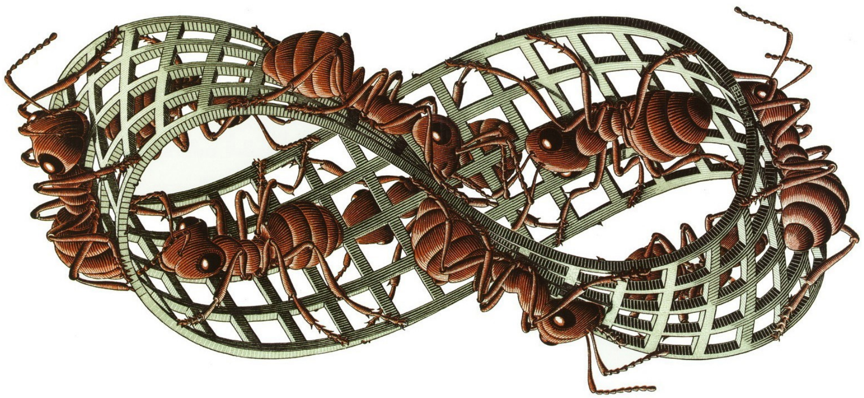
Number representation: +*Infinity* and -*Infinity*



Number representation: +*Infinity* and -*Infinity*

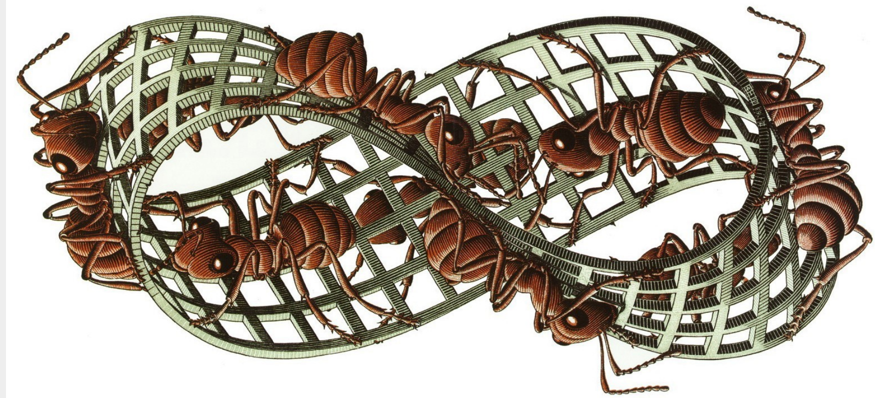
Produced by several functions/expressions and overflows. Exs (Python):

```
In [108]: import numpy, math

In [109]: c=numpy.array((0.,1000.))

In [110]: c[0]=1.0/c[0]

In [111]: c[1]=numpy.exp(c[1])

In [112]: print c
?

In [113]: print math.exp(float('-inf'))
?

In [114]: print math.atan(numpy.inf)/math.pi
?

In [115]: print c[1] > 10.0
?

In [116]: print c[1] == c[0]
?

In [117]: print math.exp(1000.)
?
```



Number representation: +*Infinity* and -*Infinity*

Produced by several functions/expressions and overflows. Exs (Python):



```
In [108]: import numpy, math

In [109]: c=numpy.array((0.,1000.))

In [110]: c[0]=1.0/c[0]

In [111]: c[1]=numpy.exp(c[1])

In [112]: print c
[ inf  inf]

In [113]: print math.exp(float('-inf'))
0.0

In [114]: print math.atan(numpy.inf)/math.pi
0.5

In [115]: print c[1] > 10.0
True

In [116]: print c[1] == c[0]
True

In [117]: print math.exp(1000.)
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
----> 1 print math.exp(1000.)
OverflowError: math range error
```

Number representation: *+NaN* and *-NaN*

Number representation: *+NaN* and *-NaN*

Number representation: *+NaN* and *-NaN*

Not a Number

Invalid results. Exs (IDL):


```
IDL> help,0./0.
<Expression>    FLOAT     =            -NaN
% Program caused arithmetic error: Floating illegal operand

IDL> help,!values.d_infinity/!values.d_infinity
<Expression>    DOUBLE    =               -NaN
% Program caused arithmetic error: Floating illegal operand

IDL> print,sqrt(-1d0)
             NaN
% Program caused arithmetic error: Floating illegal operand

IDL> print,sqrt(complex(-1d0))
(      0.00000,      1.00000)

IDL> print,!values.d_nan gt 0d0 ;NaN is not larger than anything
   0

IDL> print,!values.d_nan le 0d0 ;NaN is not smaller than anything
   0

IDL> print,!values.f_nan eq !values.f_nan ;NaN is not equal to NaN
   0
```

Just warnings, not errors

Number representation: *+NaN* e *-NaN*

Commonly used to indicate missing or nonsense data. Ex:

- Bad pixels

- Data not taken:
  - Sky area not observed
  - Magnitude not known for the object
  - Region not included in the model

Better than the common practice of picking some value like 99, -99, -1 or 0:

It is a “special” value, depending on prior knowledge.

What if not value can be special (no number makes no sense)?

Lots of software know to ignore **NaNs *in input:***

- Leave a hole in a plot.

- Ignore them when querying for maximum, minimum, mean, etc.

On most operations with NaN the result is (properly) NaN:

- Adding a number / multiplying a number to an image should not magically turn bad pixels (NaNs) into some number.
- **NaN is not 0, 1, or any other neutral element.**

Number representation: zeros (reals)

There are two zeros (+0 and -0):

Equal in comparisons, but show the difference in limits. Exs. (IDL Python):

```
IDL> print,1d0/0d0
        Infinity
% Program caused arithmetic error: Floating divide by 0


IDL> print,1d0/(-0d0)
       -Infinity
% Program caused arithmetic error: Floating divide by 0


IDL> print,0d0 eq -0d0
   1

In [133]: import numpy

In [134]: c=numpy.array((0.0,-0.0))

In [135]: print c[0] == c[1]
True

In [136]: print 1/c
[ inf -inf]
```

Just warnings, not errors.

Number representations – reals - consequences

Just like for integers, need to consider their **range**. Also their **precision limit**. Exs. (IDL):

```
IDL> print,exp(-103.)
  1.40130e-45
% Program caused arithmetic error: Floating underflow

IDL> print,exp(-104.)
% Program caused arithmetic error: Floating underflow
      0.00000
```

Just warnings, not errors.

```
In [18]: from array import array

In [19]: a=array('f',[1e9,1,1e9+1])

In [20]: print a
array('f', [1000000000.0, 1.0, 1000000000.0])

In [21]: a=array('d',[1e9,1,1e9+1])

In [22]: print a
array('d', [1000000000.0, 1.0, 1000000001.0])
```

Number representations – reals - consequences

Just like for integers, need to consider their **range**. Also their **precision limit**. Exs. (IDL):

```
IDL> print,exp(-103.)
  1.40130e-45
% Program caused arithmetic error: Floating underflow

IDL> print,exp(-104.)
% Program caused arithmetic error: Floating underflow
      0.00000
```

Just warnings, not errors.

```
In [18]: from array import array

In [19]: a=array('f',[1e9,1,1e9+1])

In [20]: print a
array('f', [1000000000.0, 1.0, 1000000000.0])

In [21]: a=array('d',[1e9,1,1e9+1])

In [22]: print a
array('d', [1000000000.0, 1.0, 1000000001.0])
```

Number representations – reals - consequences

The digits shown when a number is printed out do not necessarily correspond to its precision.

They may show more, or less than the precision, depending on how the number was printed.

Ex. (IDL):

```
IDL> print,1.0d0+1d-8
?

IDL> print,1.0d0+1d-8,format='(E22.15)'
?
```

Since the representation is binary, **only numbers that are rational in binary** (sums of powers of 2) can be represented exactly. Ex. (Python):

```
In [44]: from array import array

In [45]: a=array('f',[1.0,0.1,0.7])

In [46]: print a
?
```

Number representations – reals - consequences

The digits shown when a number is printed out do not necessarily correspond to its precision.

They may show more, or less than the precision, depending on how the number was printed.

Ex. (IDL):

```
IDL> print,1.0d0+1d-8
       1.0000000

IDL> print,1.0d0+1d-8,format='(E22.15)'
 1.000000010000000E+00
```

Since the representation is binary, **only numbers that are rational in binary** (sums of powers of 2) can be represented exactly. Ex. (Python):

```
In [44]: from array import array

In [45]: a=array('f',[1.0,0.1,0.7])

In [46]: print a
array('f', [1.0, 0.10000000149011612, 0.699999988079071])
```

Number representations – reals - consequences

In computational science, single precision is usually not enough:

- **Inverting a matrix usually does not work** (it seems singular, when it is not).

- Even if the data do not have 6 digits of precision, it may take double precision, since **consecutive operations may accumulate large errors.**

- We frequently get numbers with powers beyond ±38:
  - $h=6.62\times10^{-34}$ J×s
  - $M_\odot=1.99\times10^{33}$g
  - $M_\oplus=5.97\times10^{27}$g

- Julian dates take many digits (1 s is 1.16×10^{-5} d).
  - Ex: **2455563.024502**
  - 7 digits just to get to 1 day
  - + 5 digits to get to ~1s

- Sky coordinates are at the limit of single precision (1” takes 7 digits in decimal degrees).

Number representations – reals - consequences

Integer types have more significant digits (but smaller ranges) than reals:

- A 32 bit unsigned integer holds exactly all numbers between **0** and **4294967295** (4 giga -1 , in binary).
- A 32 bit real can hold numbers up to ~10^{38}, but **4294967295** does not exist (it would take 10 decimal digits).

Exs. (IDL):

```
IDL> a=4294967295UL

IDL> print,a,format='(I0)' & print,float(a),format='(F0)'
4294967295
4294967296.000000

IDL> print,a-25, format='(I0)' & print,float(a-25),format='(F0)'
4294967270
4294967296.000000
```

Comparing the 64 bit types:

```
IDL> a=12345678901234567890ULL

IDL> print,a,format='(I0)' & print,double(a),format='(F0)'
12345678901234567890
12345678901234567168.000000
```

Number representations – reals - consequences

Attention to comparison of real values. Exs (IDL):

```
IDL> a=dindgen(3)*!dpi

IDL> print,a
       0.0000000       3.1415927       6.2831853

IDL> print,where(a eq 3.1415927,/null)
!NULL

IDL> print,where(a eq !dpi,/null)
           1
```

Generates an array with elements 0π, 1π, 2π

No element is equal to **3.1415927**

Element **1** is equal to **!dpi**

Number representations – reals - consequences

Attention to comparison of real values. Exs (Python):

```
In [29]: import numpy,math

In [30]: a=numpy.zeros(10000)+math.pi

In [31]: b=a.sum()/a.size

In [32]: print a[0],b
3.14159265359 3.14159265359

In [33]: print a[0]==b
?

In [34]: print a[0]-b
?
```

a is an array with 100000 elements equal to **math.pi**

b is the sum of all elements of a, divided by the number of elements in a

Is **b** equal to **math.pi**?

Usually, one can only expect reals to be equal if **one is a copy of the other, and no processing was applied to them**.

Even associativity might not be true: A+(B+C) might be different from (A+B)+C.

Results may not be identical, even with the same data, with:

- Different implementations of the same algorithm.
- Different runs of the same parallel code.

Number representations – reals - consequences

Attention to comparison of real values. Exs (Python):

```
In [29]: import numpy,math

In [30]: a=numpy.zeros(10000)+math.pi

In [31]: b=a.sum()/a.size

In [32]: print a[0],b
3.14159265359 3.14159265359

In [33]: print a[0]==b
False

In [34]: print a[0]-b
5.87974113841e-13
```

a is an array with 100000 elements equal to **math.pi**

b is the sum of all elements of a, divided by the number of elements in a

Is **b** equal to **math.pi**?

No!

Usually, one can only expect reals to be equal if **one is a copy of the other, and no processing was applied to them**.

Even associativity might not be true: A+(B+C) might be different from (A+B)+C.

Results may not be identical, even with the same data, with:

- Different implementations of the same algorithm.
- Different runs of the same parallel code.

Other variable types

References / Pointers

Most languages have variables that are just references to other variables.

The meaning, occurrences and uses of references vary a lot between languages.

A **reference/pointer** is only a **link**, which points to some **target**.

A target may have several references pointing to it.

References / pointers

Caution is needed to know when a variable is a copy of another, or just another pointer to the same target. The rules are strongly language dependent.

Ex. (Python):

```
In [73]: a=9

In [74]: b=9

In [75]: id(a),id(b)
Out[75]: (7797672, 7797672)

In [76]: a=0

In [77]: id(a),id(b)
Out[77]: (7797888, 7797672)
```

References / pointers

Caution is needed to know when a variable is a copy of another, or just another pointer to the same target. Ex. (Python):

```
In [78]: a=[0,1]
In [79]: b=a

In [80]: id(a),id(b)
?

In [81]: a[0]=-1

In [82]: b
?

In [83]: b is a
Out[83]: True

In [84]: b=a[:]

In [86]: a[0]=99

In [87]: b
?

In [88]: b is a
?

In [89]: id(b),id(a)
?
```

References / pointers

Caution is needed to know when a variable is a copy of another, or just another pointer to the same target. Ex. (Python):

```
In [78]: a=[0,1]
In [79]: b=a

In [80]: id(a),id(b)
Out[80]: (40777056, 40777056)
```

Both **a** and **b** point to the same target.

```
In [81]: a[0]=-1
```

Doing some change to **a**'s target.

```
In [82]: b
Out[82]: [-1, 1]
```

The change is seen in **b**'s target (since it is the same as **a**'s target.

```
In [83]: b is a
Out[83]: True

In [84]: b=a[:]
```

Now **b** is created differently.

```
In [86]: a[0]=99
```

a's target is edited.

```
In [87]: b
Out[87]: [-1, 1]
```

b's target is unnafected.

```
In [88]: b is a
Out[88]: False

In [89]: id(b),id(a)
Out[89]: (37221368, 40777056)
```

References / pointers

Caution is needed to know when a variable is a copy of another, or just another pointer to the same target. Ex. (Python):

```
In [73]: a=9

In [74]: b=9

In [75]: id(a),id(b)
?

In [76]: a=0

In [77]: id(a),id(b)
?

In [76]: a,b
?
```

References / pointers

Caution is needed to know when a variable is a copy of another, or just another pointer to the same target. Ex. (Python):

```
In [73]: a=9

In [74]: b=9

In [75]: id(a),id(b)
Out[75]: (7797672, 7797672)

In [76]: a=0

In [77]: id(a),id(b)
Out[77]: (7797888, 7797672)

In [76]: a,b
Out[76]: (0, 9)
```

→ Both **a** and **b** point to the same target. (re: In [75])

→ Now **a**'s target is different. (re: In [77])

Other variable types

Can Integers / Reals / Strings do everything?

No!

What if I need to carry around a lot of information?

Ex: When processing observations, the program needs to know, for each image:

- File name
- Number of sources found in the image
- Coordinates (RA/Dec) of each source in the image
- Number of point sources found in the image
- Number of moving sources found in the image
- Image quality measurements
- Magnitude of each source in the image
- Observation date/time
- Instrument
- Exposure time
- ....

Other variable types

Then carrying around variables is cumbersome. Ex (Python):

```
for i in range(len(file)):
   do_fancy_processing(file=file[i],nsources=nosources[i],ras=ra[i],
decs=decs[i],npoint=npoint[i],nmoving=nmoving[i],fwhm=fhwm[i],
mags=mags[i],obsdate=obsdate[i],....)
```

Filtering the data is even worse:

```
w=numpy.where(nsources > 0)
file=file[w]
nsources=nsources[w]
ras=ras[w]
decs=decs[w]
npoint=npoint[w]
nmoving=nmoving[w]
fwhm=fwhm[w]
mags=masg[w]
obsdate=obsdate[w]
.....
```

And don't you dare forget to do this to one of the 49 variables!

There must be a better way...

Other variable types - structures

A **structure*** is a compound type.

- Contains several fields
- Each field is a variable, of any type (even structure)
- Each field is identified by a name

Ex:

*Not to be confused with *data structure*, which means *a way to organize data* (i.e., arrays, lists, dictionaries, trees, etc.)

Other variable types - structures

Ex: (Python)

```
In [62]: import numpy as np
In [63]: obs=np.zeros(3,dtype=[('file','a256'),('nsources','i8'),
('ras',object),('decs',object),('npoint','i8'),('nmoving','i8'),
('fwhm','f8'),('mags',object),('obsdate','a22')])

In[64]:obs[0]=('something.fits',7,np.zeros(7,'i8'),np.zeros(7,'i8'),5,2,0.58,
np.zeros(7,'i8'),'2014-01-17-17:43:26.34')
In [65]: obs['fwhm']=0.58,0.98,0.73

In [66]: obs[0]
Out[66]: ('something.fits', 7, [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0],
5, 2, 0.58, [0, 0, 0, 0, 0, 0, 0], '2014-01-17-17:43:26.34')

In [68]: obs['fwhm']
Out[68]: array([ 0.58,  0.98,  0.73])

In [69]: obs['fwhm'][0]
Out[69]: 0.57999999999999996

In [87]: w=np.where(obs['nsources'] > 0)

In [89]: obs=obs[w]

In [93]: len(obs)
Out[93]: 2

In [94]: for iobs in obs:
  do_fancy_processing(iobs)
```

Other variable types - structures

Ex: (IDL)

```
IDL>observation={file:'something.fits',nsources:1701,ras:dblarr(1701),decs:db
larr(1701),npoint:1208,nmoving:7,fwhm:0.58d0,mags:dblarr(1701),obsdate:'2014-
01-17-17:43:26.34'}
IDL> observations=replicate(observation,172)
IDL> help,observations
OBSERVATIONS    STRUCT    = -> <Anonymous> Array[172]
IDL> help,observations[0]
** Structure <de2578>, 9 tags, length=40880, data length=40870, refs=3:
   FILE            STRING    'something.fits'
   NSOURCES        INT           1701
   RAS             DOUBLE    Array[1701]
   DECS            DOUBLE    Array[1701]
   NPOINT          INT           1208
   NMOVING         INT              7
   FWHM            DOUBLE         0.58000000
   MAGS            DOUBLE    Array[1701]
   OBSDATE         STRING    '2014-01-17-17:43:26.34'

IDL> print,observations[0].nsources
    1701

IDL> help,observations.nsources
<Expression>    INT       = Array[172]

IDL> foreach observation,observations do do_fancy_processing(observation)

IDL> observations=observations[where(observations.nsources gt 0)]
```

Other variable types - structures

Common uses for structures (and arrays of structures):

- Group together a lot of variables that are related:
  - Information on observations, files, models, objects (previous example)
  - All the many inputs and outputs of a complicated program.
  - Represent tables of data from files / databases (each row is a structure). Ex. (Python):

```
In [11]: f=pyfits.open('dr10_Field_pfpenteado.fit')

In [12]: f
Out[12]:
[<pyfits.hdu.image.PrimaryHDU at 0x20b4550>,
 <pyfits.hdu.table.BinTableHDU at 0x20ba950>]
In [13]: table=f[1].data
In [14]: table['mjd_u']
Out[14]:
array([ 51075.23486904,  51075.23528363,  51075.23569821, ...,
        55153.16244544,  55153.16286008,  55153.16327471])
In [15]: table[0]
Out[15]: (1237645876861272064, 94, 301, 1, 11, 3, 51075.234869040039,
51075.236527379973, 51075.233210700004, 51075.235698209959, ...
In [16]: table.names
Out[16]:
['fieldID',
 'run',
 'rerun',
 'camcol',
...
```

Other variable types - objects

Objects are the next step in complexity for types:

- **Integers**
  - One value, an integer with a simple binary coding.
- **Reals**
  - One value, represented by a complicated standard (sign, exponent, fraction, special values).
- **Structures**
  - Several values (fields) in a group, of varied types, identified by names.
  - **Code must know specifically what to do with each field. If they receive a structure of a different type, or with inconsistent data, they may end up doing the wrong thing.**
- **Objects**
  - Structures (where the data is stored) + code (which operates on the object's data)
  - Data is kept inside the object. Only the object's routines have access to the data.
  - The previous types are just static data stores. They do nothing. Objects are "variables that do stuff".

Other variable types - objects

What are objects for? Why would I want one?

- Procedural (non-object) programming:
  - ➔ There are a lot of variables around, of many different types.
  - ➔ The programmer must know what each variable means, and what to do with them.
  - ➔ The programmer must carry all associated variables around, and keep them valid. Ex:

```
IDL> help,observations[0]
** Structure <de2578>, 9 tags, length=40880, data length=40870, refs=3:
   FILE             STRING    'something.fits'
   NSOURCES         INT           1701
   RAS              DOUBLE    Array[1701]
   DECS             DOUBLE    Array[1701]
   NPOINT           INT           1208
   NMOVING          INT              7
   FWHM             DOUBLE        0.58000000
   MAGS             DOUBLE    Array[1701]
   OBSDATE          STRING    '2014-01-17-17:43:26.34'
```

Must be kept consistent. It is up to the programmer to make sure `nsources, ras, decs and mags` match.

  - ➔ **The programmer calls routines, giving variables to them. These routines must know what to do with whatever variables they are given. Ex:**

```
a=mean(b)
```

What is the type of **`b`**? (array? list? dictionary?) Does the function (**`mean`**) know what to do with it?

What happens if I make up a new type? Do I have to change the function (**`mean`**) so that it can handle the new type?

Other variable types - objects

What are objects for? Why would I want one?

- Object-oriented programming (OOP)
  - ➔ There are few variables visible, of different types.
  - The objects contain a lot of variables inside them, but these are not visible.
  - The programmer asks the variables to do things.
  - The code that does these things lives inside the variable's type definition, so it knows how data is organized, and what to do with it.

Ex:

- Procedural programming:

```
a=mean(b)
```

This is a call to a function called **`mean`**, **which is global** will have to figure out what to do with the variable (`b`).

- Object-oriented programming:

```
a=b.mean()
```

This is a call to the function called **`mean`**, which belongs to the type of the variable (`b`), whatever that type is. If `b` is an array, array's **`mean`** will be called. If `b` is a list, list's **`mean`** will be called. There is no risk the function will get the wrong type of variable.

Objects x structures

A passive variable (structure) does nothing. The programmer must know the variables,

know what to do with them, and do it.

Objects x structures

A passive variable (structure) does nothing. The programmer must know the variables,

know what to do with them, and do it.

An active variable (object), however, contains all the necessary variables, and knows what to do with them. The programmer just has to turn it on (call the object's functions):

Objects x structures

A passive variable (structure) does nothing. The programmer must know the variables,

know what to do with them, and do it.

An active variable (object), however, contains all the necessary variables, and knows what to do with them. The programmer just has to turn it on (call the object's functions):

Objects - nomenclature

- *Objects* are variables, and are ***instances*** of ***classes***.

- The **class** is an object's **type**.

- An instance is an exemple of a type::
  - **2** is an **instance** of the type **integer**.
  - **1.0** is an **instance** of the type **real**

- Objects are structures, with added routines (***methods***) which operate on them.

- Classes have ***inheritance***:
  - A new class (**ClassB**) can be made by inheriting from another class (**ClassA**).
  - **ClassA** is a ***superclass*** of **ClassB**.
  - Every object of **ClassB** is also an object of **ClassA**, and inherits all of **ClassA**'s characteristics. It may add characteristics (data, methods), or change those it inherited.

More uses for objects

Objects contain everything they need, and can produce data on demand.

Ex: Reading and using a lot of different data from a complicated file.

- Procedural-style:

```
complicated_file_reader(file='complicated_file.fits',image=image,columns=co
lumns,rows=rows,exptime=exptime,obsdate=obsdate,instrument=instrument,
targetname=targetname,...)

#(do a bunch of stuff with all those variables)
```

- Object-oriented style:

```
f=pyfits.open('complicated_file.fits')

make_pretty_figure(data=f[0].data,title=f[0].header['targname'])

for column in range(f[0].header['NAXIS1']):
   for row in range(f[0].header['NAXIS2']):
      a=f[0].data[column,row]/f[0].header['EXPTIME']
      if f[0].header['INSTRUME'] == 'ACS':
        #do some stuff
```

More uses for objects

Objects contain everything they need, and can produce data on demand.

Ex (Python): A FITS file is read into an object, that knows how to do a lot of stuff:

```
In [14]: import pyfits
In [15]: f=pyfits.open('something.fits')
In [16]: dir(f)
Out[16['_HDUList__file','__add__','__class__','__contains__','__setslice__'
,'__sizeof__','__str__','__subclasshook__','append','close','count','extend
','fileinfo','filename','flush','update_extend','verify','writeto',...]
In [18]: f[0]
Out[18]: <pyfits.hdu.image.PrimaryHDU at 0x1c9ad10>
In [19]: f[0].header['EXPTIME']
Out[19]: 49266.0
In [20]: f[0].data
Out[20]:
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]], dtype=float32)

In [21]: f.writeto('someotherfile.fits')
```

More uses for objects

Objects contain the data, and keep it consistent. The programmer cannot mess with it.

Ex (Python):

```
In [34]: import numpy,pyfits
In [35]: f=pyfits.open('something.fits')
In [36]: f[0].header['NAXIS1']
Out[36]: 10000
In [37]: f[0].header['NAXIS2']
Out[37]: 10000
In [38]: f[0].data
Out[38]:
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]], dtype=float32)

In [39]: f[0].data=numpy.array([[1,7],[8,3],[5,9]])
In [40]: f[0].header['NAXIS1']
?
In [41]: f[0].header['NAXIS2']
?
```

More uses for objects

Objects contain the data, and keep it consistent. The programmer cannot mess with it.

Ex (Python):

```
In [34]: import numpy,pyfits
In [35]: f=pyfits.open('something.fits')
In [36]: f[0].header['NAXIS1']
Out[36]: 10000
In [37]: f[0].header['NAXIS2']
Out[37]: 10000
In [38]: f[0].data
Out[38]:
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]], dtype=float32)

In [39]: f[0].data=numpy.array([[1,7],[8,3],[5,9]])
In [40]: f[0].header['NAXIS1']
Out[40]: 2
In [41]: f[0].header['NAXIS2']
Out[41]: 3
```

But be careful! Do not assume that the class is perfect, and keeps everything consistent all the time. It depends on how careful the class' writer was.

More uses for objects

The methods in a class define what happens if you use an operator with an object of that class (operator overloading).

Ex (Python):

```
In [56]: from array import array

In [57]: import numpy

In [58]: aa=array('i',(0,1))

In [59]: ab=array('i',(0,-1))

In [60]: aa+ab
?

In [61]: na=numpy.array((0,1),dtype='i')

In [62]: nb=numpy.array((0,-1),dtype='i')

In [63]: na+nb
?
```

More uses for objects

The methods in a class define what happens if you use an operator with an object of that class (operator overloading).

Ex (Python):

```
In [56]: from array import array

In [57]: import numpy

In [58]: aa=array('i',(0,1))

In [59]: ab=array('i',(0,-1))

In [60]: aa+ab
Out[60]: array('i', [0, 1, 0, -1])

In [61]: na=numpy.array((0,1),dtype='i')

In [62]: nb=numpy.array((0,-1),dtype='i')

In [63]: na+nb
Out[63]: array([0, 0], dtype=int32)
```

Be careful! It is up to a class' writer to decide the meaning of the operators.

Number representations - consequences

What is wrong with this? (Python)

```
def stefanboltzmann(j):
  sigma=5.670400e-8 #Js^-1m^-2K^-4
  return (j/sigma)**(1/4)

print stefanboltzmann(6.3200984e7)
?
```

(IDL)

```
function stefanboltzmann,j
  sigma=5.670400e-8 ;Js^-1m^-2K^-4
  return, (j/sigma)^(1/4)
end

print, stefanboltzmann(6.3200984e7)
end
?
```

Number representations - consequences

What is wrong with this? (Python)

```
def stefanboltzmann(j):
  sigma=5.670400e-8 #Js^-1m^-2K^-4
  return (j/sigma)**(1/4)

print stefanboltzmann(6.3200984e7)
1.0
```

(IDL)

```
function stefanboltzmann,j
  sigma=5.670400e-8 ;Js^-1m^-2K^-4
  return, (j/sigma)^(1/4)
end

print, stefanboltzmann(6.3200984e7)
end
      1.00000
```

Number representations - consequences

What is wrong with this? (Python)

```
import numpy as np
def comparecolors(color1,color2):
  return np.amax(np.abs(color1-color2))

color1=np.array((200,190,0),dtype='u1')
color2=np.array((198,190,0),dtype='u1')
color3=np.array((201,190,0),dtype='u1')



print comparecolors(color1,color2)
print comparecolors(color1,color3)
?
?
```

(IDL)

```
function comparecolors,color1,color2
  return,max(abs(color1-color2))
end

print,comparecolors([200B,190B,0B],[198B,190B,0B])
print,comparecolors([200B,190B,0B],[201B,190B,0B])
end
?
?
```

Number representations - consequences

What is wrong with this? (Python)

```
import numpy as np
def comparecolors(color1,color2):
  return np.amax(np.abs(color1-color2))

color1=np.array((200,190,0),dtype='u1')
color2=np.array((198,190,0),dtype='u1')
color3=np.array((201,190,0),dtype='u1')



print comparecolors(color1,color2)
print comparecolors(color1,color3)
2
255
```

(IDL)

```
function comparecolors,color1,color2
  return,max(abs(color1-color2))
end

print,comparecolors([200B,190B,0B],[198B,190B,0B])
print,comparecolors([200B,190B,0B],[201B,190B,0B])
end
   2
 255
```

Real questions, from the IDL newsgroup

1)
This may be a stupid question, but I really want to know why.
Please, see below and explain. Thanks.

```
IDL> print, 132*30
    3960
IDL> print, 132*30*10
  -25936
```

2)
There's something I can not explain to myself, so maybe someone can enlighten me?

```
IDL> print, fix(4.70*100)
        469
```

To try and find where the problem is, we tried the following lines:

```
  IDL> a = DOUBLE(42766.080001)
  IDL> print,a,FORMAT='(F24.17)'
      42766.07812500000000000
```

As you see, the number we get out isn't the same as the number we entered.

3)
I have a problem related to float-point accuracy
If I type in: 50d - 1d-9, I get 50.000000
And here lies my problem, I'm doing a numerical simulation where such an arithmetic is common place, and as a result i get a lot or errors. I know for example, that if i simply type
print, 50d - 1d-9, format = '(f.20.10)' , i'll get:
49.9999999990
But how can I convince IDL to do it on its own during computations?

Real questions, from the IDL newsgroup

4)
Hi guys,
IDL> ***print,((10^5)/(exp(10)*factorial(5)))***
The actual result of the above line is 0.0378332748
But when we run it in IDL we get the result as -0.011755556

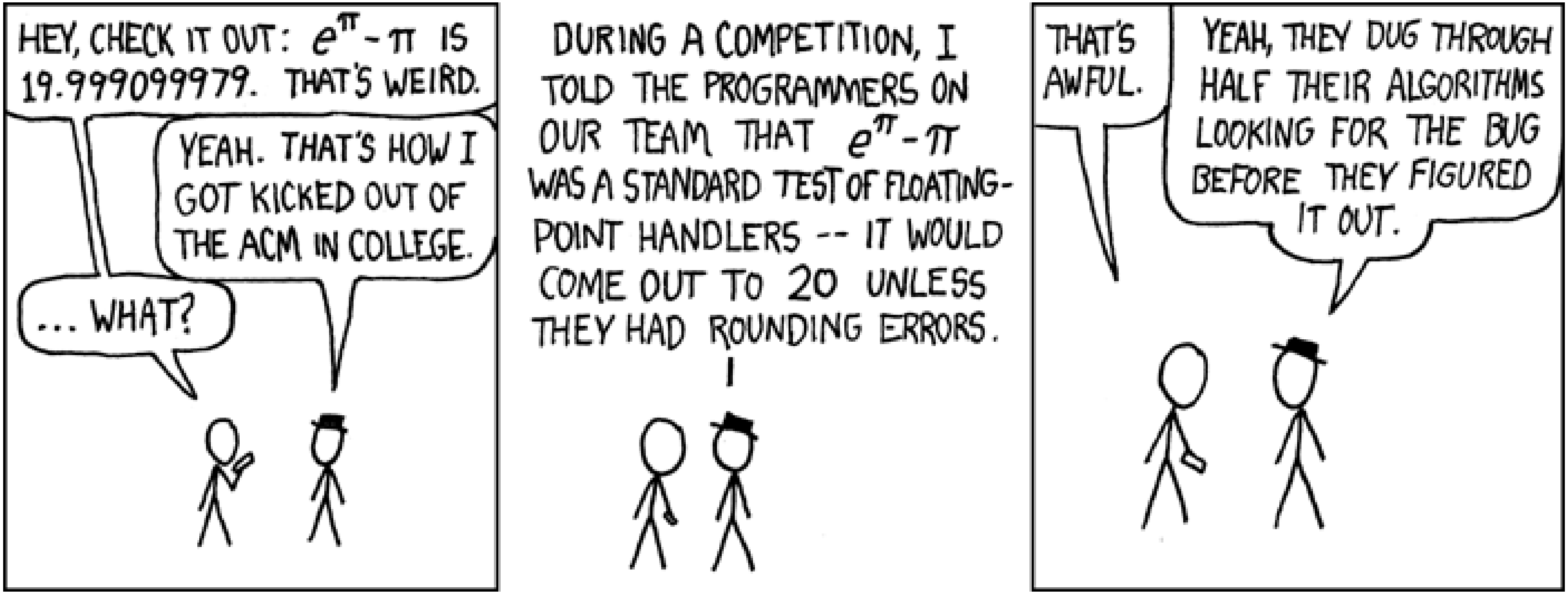
5)
I ran into a number transformation error yesterday that is still confusing me this morning. The problem is that the number ***443496.984*** *is being turned into the number* ***443496.969*** *from basic assignments using Float() or Double(), despite the fact that even floats should easily be able to handle a number this large (floats can handle "±10^38, with approximately six or seven decimal places of significance").*

Some References

Help! The sky is falling!
http://www.dfanning.com/math_tips/sky_is_falling.html

What every programmer should know about floating-point arithmetic
or
Why don't my numbers add up?
http://floating-point-gui.de/

What every computer scientist should know about floating-point arithmetic
http://docs.sun.com/source/806-3568/ncg_goldberg.html



http://www.xkcd.org/217