

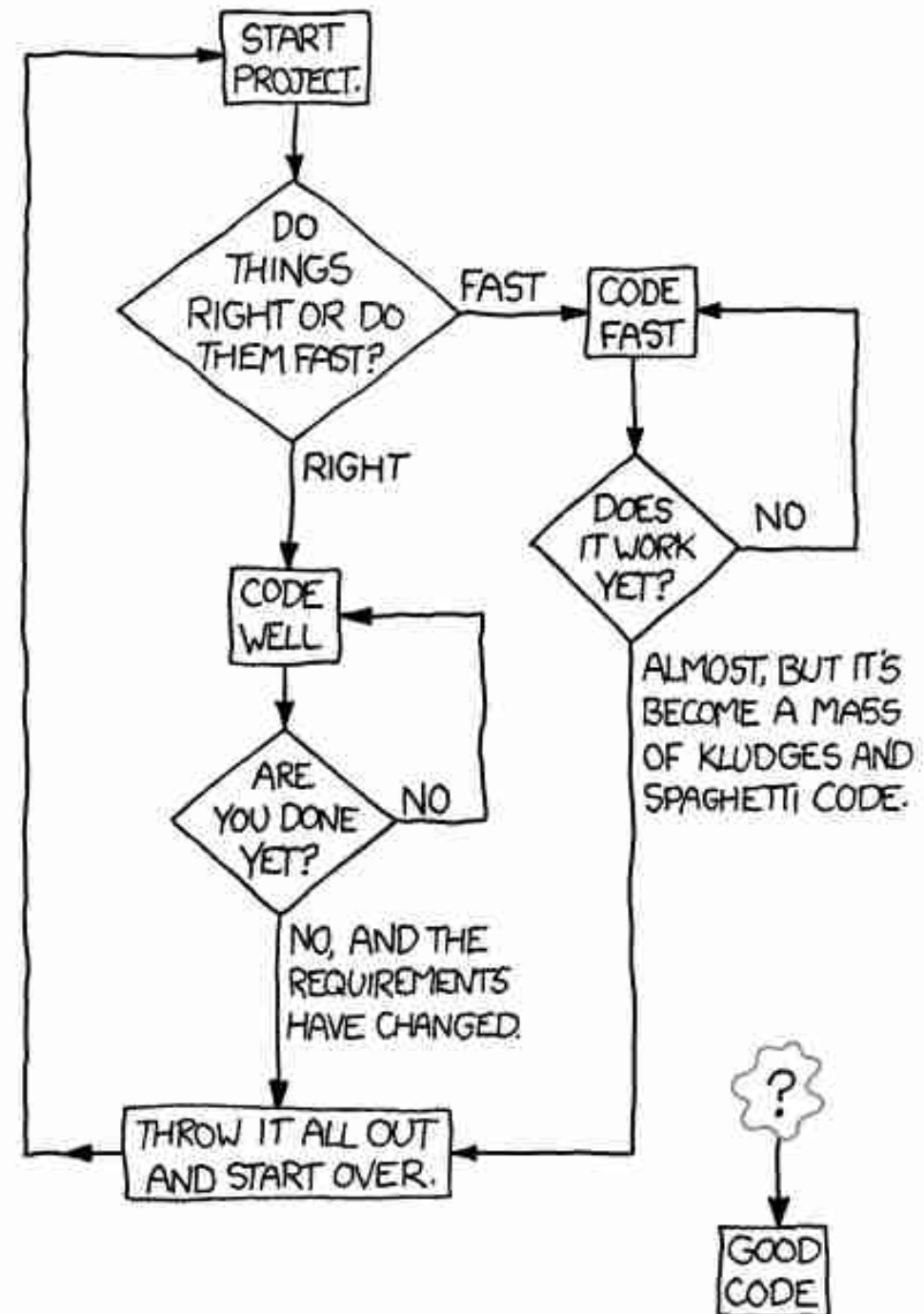
# Programação em astronomia: indo além de loops e prints

## 1 - Linguagens

Paulo Penteado

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

HOW TO WRITE GOOD CODE:



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

# Programa

1 – Slides em [http://www.ppenteado.net/pea/pea01\\_linguagens.pdf](http://www.ppenteado.net/pea/pea01_linguagens.pdf)

- Motivação
- Tópicos abordados
- Tópicos omitidos
- Opções e escolha de linguagens
- Uso de bibliotecas
- Referências

2 – Slides em [http://www.ppenteado.net/pea/pea01\\_organizacao.pdf](http://www.ppenteado.net/pea/pea01_organizacao.pdf)

- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing

3 – Slides em [http://www.ppenteado.net/pea/pea02\\_variaveis.pdf](http://www.ppenteado.net/pea/pea02_variaveis.pdf)

- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

# Programa

- 4 – Slides em [http://www.ppenteado.net/pea/pea03\\_containers.pdf](http://www.ppenteado.net/pea/pea03_containers.pdf)
  - Contêiners
  - Arrays
  - Listas
  - Mapas
  - Outros contêiners
  - Vetorização
  - Escolha de contêiners
  
- 5 – Slides em [http://www.ppenteado.net/pea/pea04\\_strings\\_io.pdf](http://www.ppenteado.net/pea/pea04_strings_io.pdf)
  - Strings
  - Expressões regulares
  - Arquivos

# Motivação

Grande parte da pesquisa em astronomia\* hoje necessita de programação

Tanto observações como modelos geram grandes quantidades de dados, que precisam de

- Organização
- Processamento
- Visualização

Algoritmos, para qualquer propósito, precisam ser organizados, documentados e testados, para serem

- Confiáveis
- Reprodutíveis (*“como era mesmo o modelo que gerou aquele espectro do artigo?”*)
- Reusáveis (*“o que faz e como se usa aquele programa escrito 8 meses atrás?”*)
- Práticos

Cursos de graduação e pós-graduação são tradicionalmente omissos em programação:

- Entre dezenas de disciplinas obrigatórias, tipicamente 0 a 2 tentam ensinar programação
- Estas poucas são tipicamente deficientes:
  - Linguagem imposta, sem escolha (Fortran 77 sendo comum)
  - Limitadas a expressões matemáticas, loops, ifs e prints
  - Omitem: escolha de linguagem, organização, documentação, testes, armazenamento de dados, visualização, e muitas das ferramentas importantes das linguagens e suas bibliotecas.

\*Ciências computacionais como astronomia, geofísica, ciências atmosféricas, física, e provavelmente outras; aqui serão abreviadas apenas como “astronomia”, para não generalizar excessivamente para todas as ciências computacionais.

# Motivação

Programação ocupa grande parte do tempo de trabalho dos astrônomos.

Mas durante sua formação esta tipicamente recebe muito menos esforço que matemática, física e astronomia.

Resultados:

- Programar toma muito mais tempo, e gera muito mais sofrimento que o necessário.  
→ Especialmente relevante para os que *não gostam* de programar.
- Os resultados são muito mais limitados, frágeis e difíceis de reproduzir que poderiam ter sido pelo trabalho investido neles.
- Reaproveitamento e expansão do trabalho anterior são dificultados.
- Códigos são menos eficientes.
- Códigos são de difícil verificação e reuso (por outros e pelos próprios autores).
- Mitos se propagam.
- Práticas ruins se propagam.

# Motivação

```
C:\lab>
f?? -o
data.exe
```

```
>
>
```

**... ERROR**

```
...why scientific programming does not
compute
```

```
>
```

**11/21/10 MERALI**

```
>
>
```

## ...SCIENTISTS AND THEIR SOFTWARE

A survey of nearly 2,000 researchers showed how coding has become an important part of the research toolkit, but it also revealed some potential problems.

> **45%** said scientists spend more time today developing software than five years ago."

> **38%** of scientists spend at least one fifth of their time developing software.

> Only **47%** of scientists have a good understanding of software testing.

> Only **34%** of scientists think that formal training in developing software is important.

# Motivação

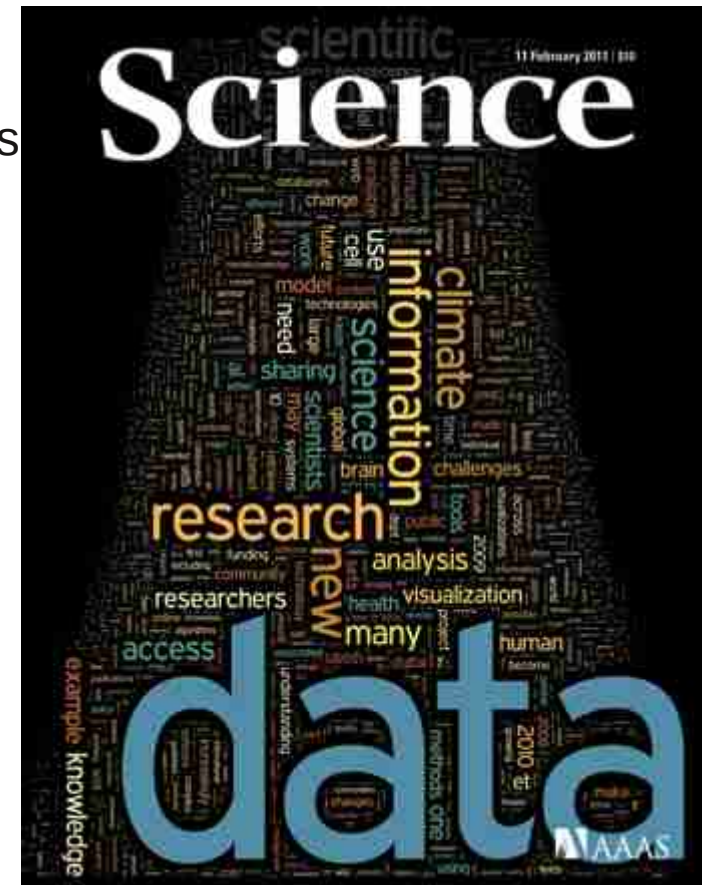
Ciência da computação é a de evolução mais rápida hoje:

- Novos métodos e paradigmas em escalas < 10 anos.
- Torna os códigos mais robustos, eficientes e simples, com resultados melhores.

Esta evolução é necessária, para lidar com os problemas mais complexos, e com a crescente quantidade de dados.

Assunto da edição especial da Science (11/fev/2011):

<http://www.sciencemag.org/site/special/data/>



Não fazer proveito da evolução é sofrer desnecessariamente:

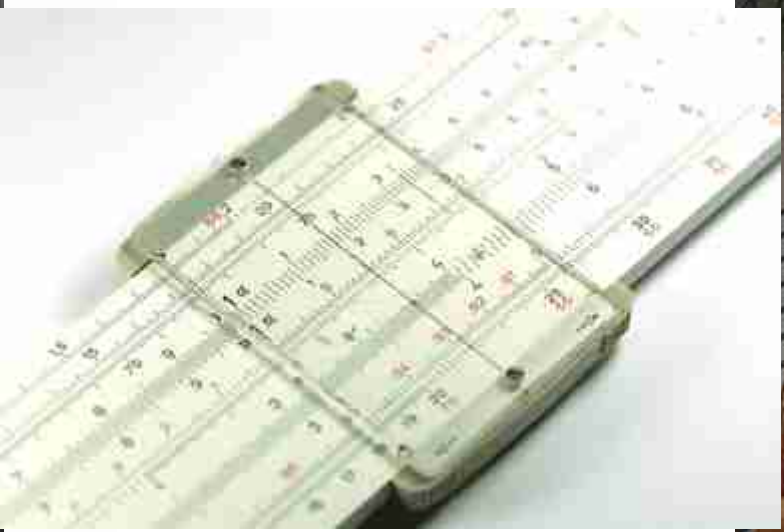
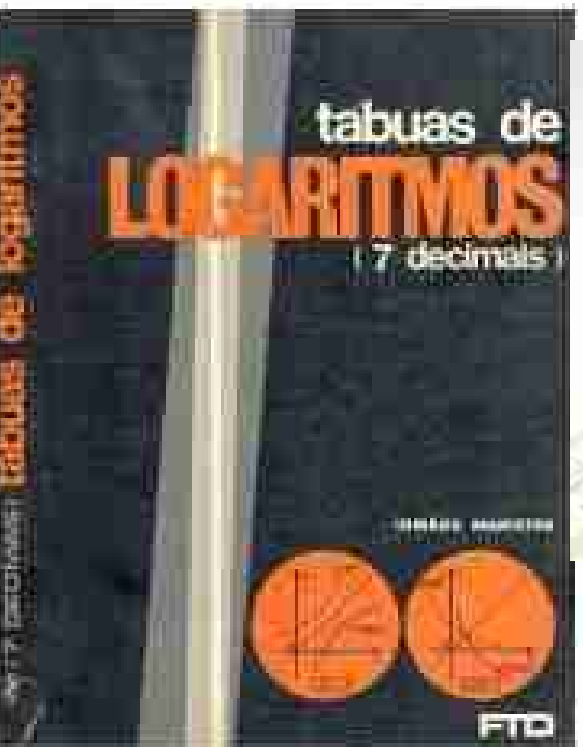
*“We discuss what hampers the rate of scientific progress in our exponentially growing world (...) The reason for this lies in the education of astronomers lacking basic computer science aspects crucially important in the data intensive science era.”*

Chilingarian e Zolotukhin, *The True Bottleneck of Modern Scientific Computing in Astronomy*

<http://arxiv.org/abs/1012.4119>

# Motivação

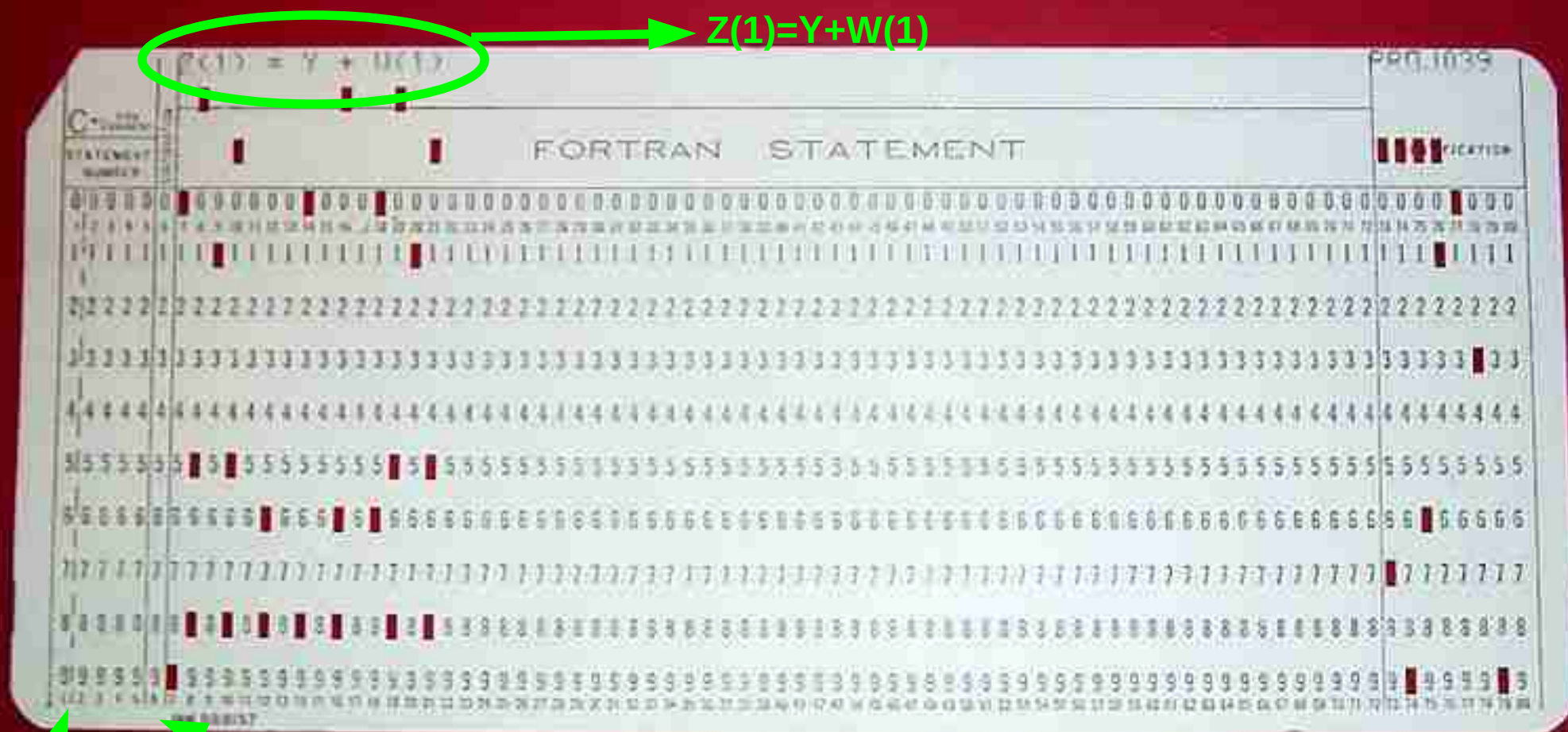
Você não usa:



Então por que programar como um programador iniciante em 1970?

# Motivação

Quem usa Fortran em formato fixo (como em F77, para cartões perfurados) deve reconhecer:



Marcador de continuação

numeração

80 colunas

Marcador de comentários

72 colunas

8 ignoradas

Por que continuar contando colunas?

# Motivação

Deficiências de educação em programação em astronomia semelhantes às deficiências em educação em estatística (limitada à estatística até o século XIX).

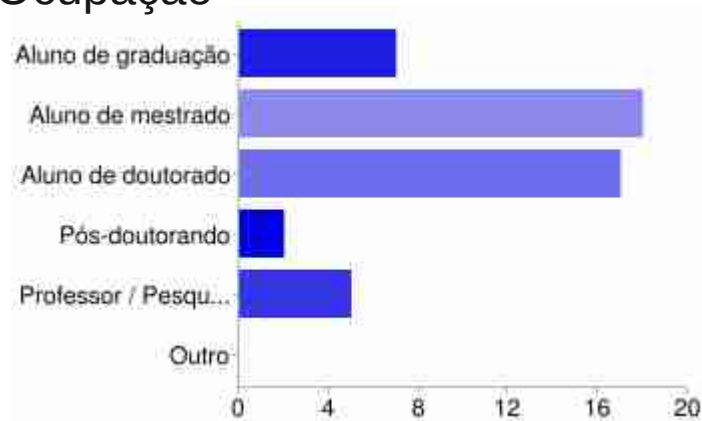
Além de boa instrução inicial é necessário haver atualização: há mudanças grandes em 5 anos.

Exemplos de algumas novidades relevantes:

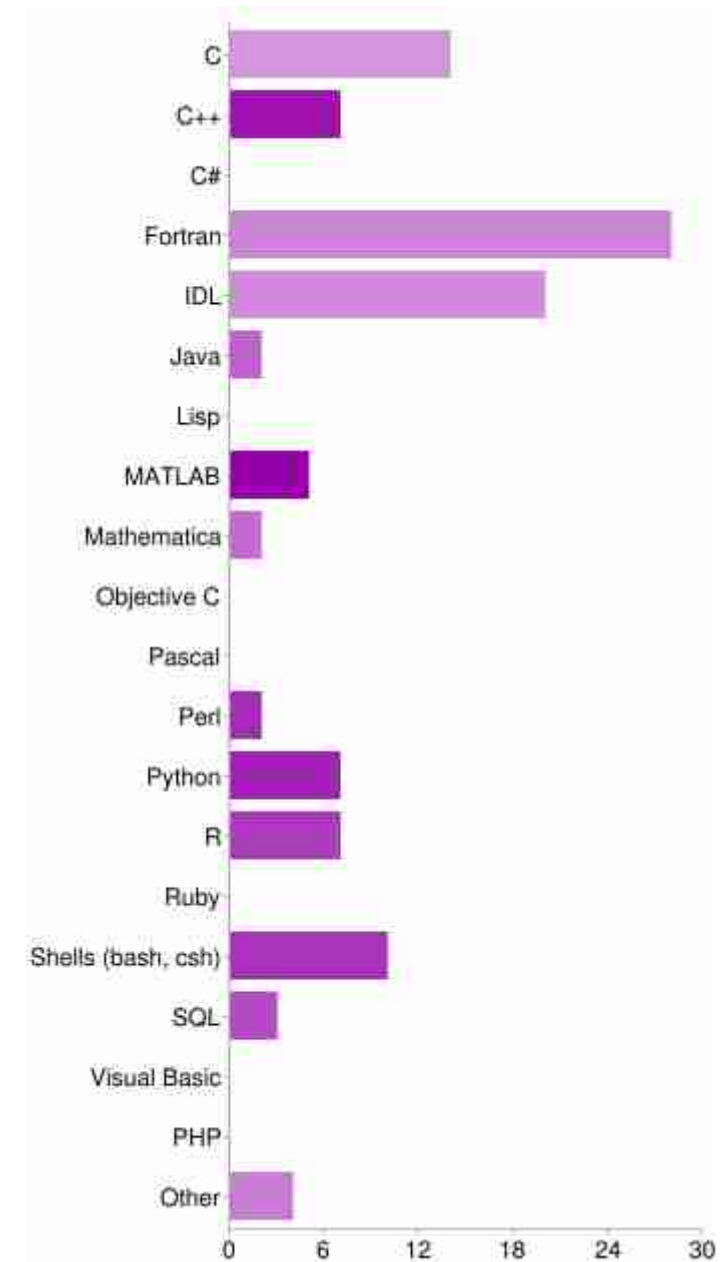
1994	MPI
1995	MySQL
1996	Java 1.0
1998	primeiro C++ padronizado
2000	Python 2
2000	OpenMP 2.0
2000	Perl 5.6
2000	R 1.0
2003-2010 (?)	Fortran 2003 (adicionando “objetos”)
2004	Java 1.5
2005	R 2.1
2005	NumPy
2007	CUDA 1.0 (para GPUs NVIDIA)
2008	OpenCL (para qualquer GPU)
2008	Python 3
2008	Objetos em MATLAB
2008	OpenMP 3.0
2010-?	Fortran 2008
2010	IDL 8.0
2010	NumPy em Python 3
2011	C++11

# Resultados da pesquisa anônima entre os inscritos (49)

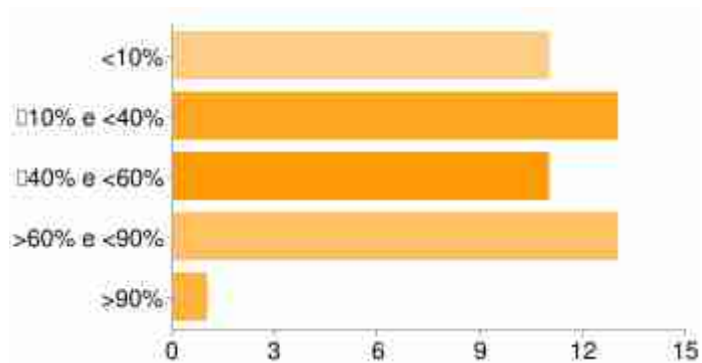
## Ocupação



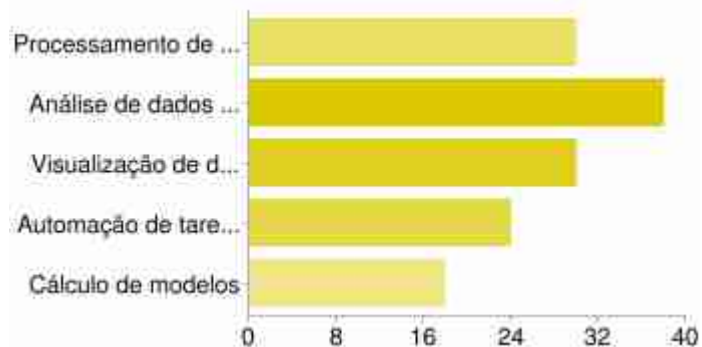
## Principais linguagens usadas



## Tempo gasto programando

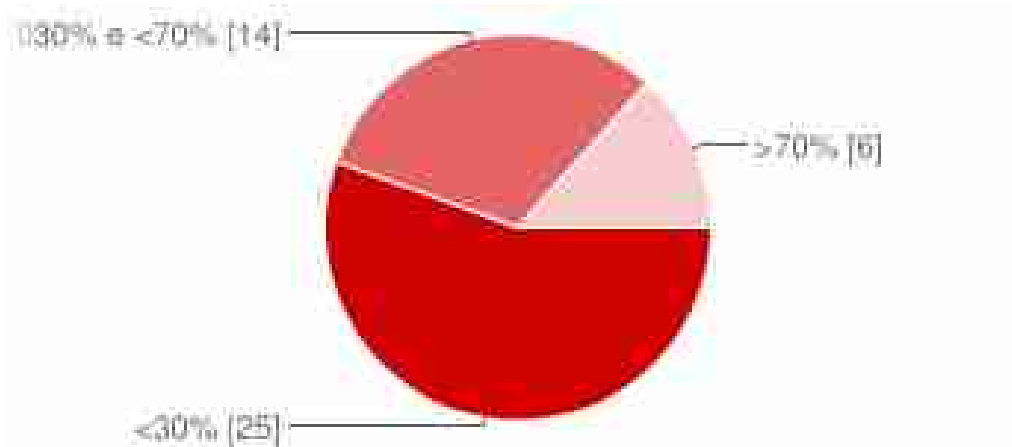


## Áreas de aplicação

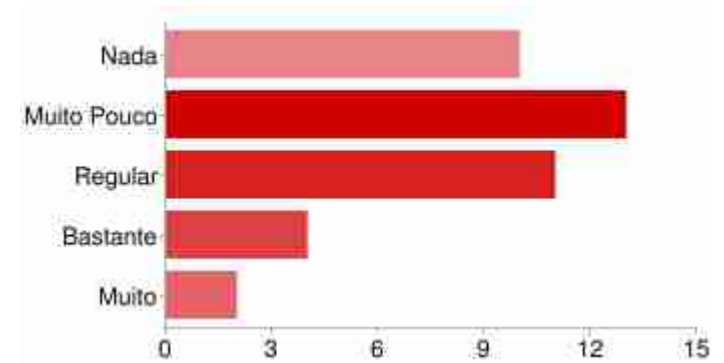


# Resultados da pesquisa anônima entre os inscritos (49)

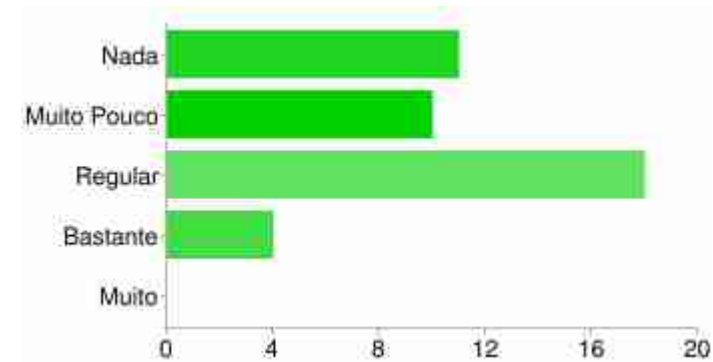
Código próprio (ou alterado, de outros)



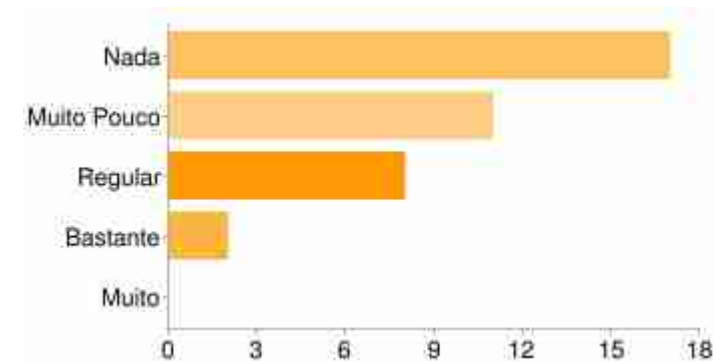
IDL



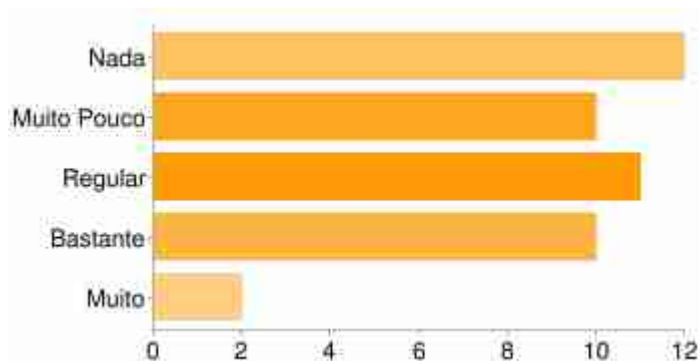
C



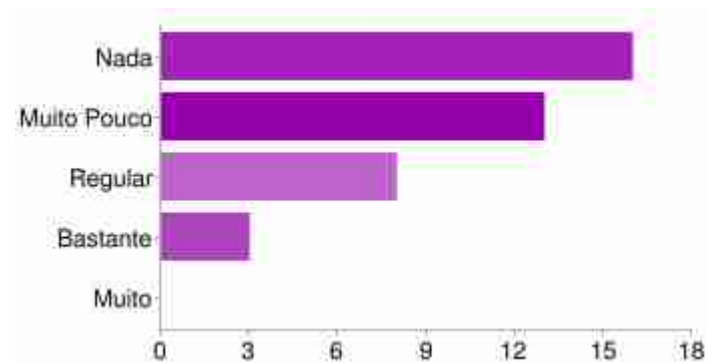
Shells



Fortran



C++



Todas as outras linguagens tiveram respostas predominantemente “Nada” e “Muito Pouco”

# Tópicos abordados

Introdução aos pontos básicos (necessários a todos) mais comumente negligenciados:

- Motivação
- Tópicos abordados
- Tópicos omitidos
- Opções e escolha de linguagens
- Uso de bibliotecas
- Referências
  
- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing
  
- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

# Tópicos abordados

Introdução aos pontos básicos (necessários a todos) mais comumente negligenciados:

- Contêiners
  - Arrays
  - Listas
  - Mapas
  - Outros contêiners
  - Vetorização
  - Escolha de contêiners
- 
- Strings
  - Expressões regulares
  - Arquivos

# Tópicos omitidos

Muito importantes para muitas aplicações, mas como não relevantes a todas:

- Bancos de dados (esp. SQL)
- Acesso a rede
- Paralelização de dados e de tarefas
- Acesso direto a hardware
- Compilação, makefiles
- Segurança (*security*, não *safety*)
- Interação entre linguagens diferentes
- Scripts
- Conteúdo web (esp. HTML, CSS, PHP, JavaScript)
- Algoritmos específicos (processamento de dados, estatística, matemática, uso geral)
- Visualização
- Interfaces gráficas

Tratamento de exceções será também omitido, por ser muito variável entre linguagens.

# Opções e escolha de linguagens

A primeira decisão relevante, e de grande importância, com frequência é ignorada:

**Que linguagem usar? - Qual é a melhor?**

**Mito:** *“Não importa a linguagem, são todas equivalentes”*

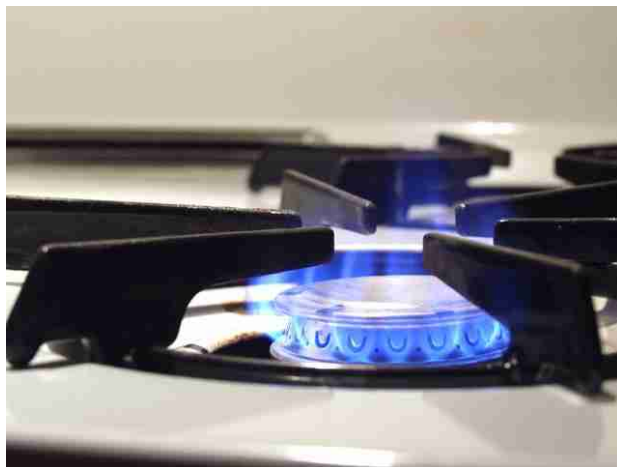
# Opções e escolha de linguagens

A primeira decisão relevante, e de grande importância, com frequência é ignorada:

**Que linguagem usar? - Qual é a melhor?**

**Mito:** *“Não importa a linguagem, são todas equivalentes”*

Não são equivalentes, assim como não são equivalentes



Dá para cozinhar qualquer coisa tendo só uma churrasqueira? Dá para fazer arroz usando só um forno a gás? Talvez, mas com muito sofrimento desnecessário.

# Opções e escolha de linguagens

Infelizmente, não existe “*a melhor linguagem*”.

Não há uma que seja a melhor para qualquer uso geral (ou mesmo para qualquer uso comum em astronomia), porque os problemas a resolver são variados, e as linguagens têm propósitos variados.

“*A melhor*” depende **principalmente (não exclusivamente)** do problema específico.

- Mesmo para um dado problema não há só uma solução.

Não há (em geral) uma “resposta certa” sobre escolha de linguagem / algoritmo / contêiner.

**Um problema comum é a imposição de uma solução, sem dar a opção de escolha, ou assumir um caminho, sem considerar o que pode ser melhor.**

Cada linguagem foi desenhada com um objetivo, e se adequa melhor a alguns problemas.

Exemplos de linguagens mais adequadas (mais detalhes adiante):

- Arrays em várias dimensões – IDL, Python+NumPy
- Portabilidade, em múltiplas plataformas – Java, IDL
- Strings e expressões regulares – Perl, Python, IDL, Java, C++
- Visualização – IDL, Python, R
- Algoritmos simples e computacionalmente pesados – C++, Fortran
- Estatística (não é só variância, desvio padrão, MQ e  $\chi^2$ ) – R

# Opções e escolha de linguagens

**Não é possível escolher uma única para aprender uma vez e nunca mais estudar algo novo:**

- Problemas diferentes vão pedir ferramentas diferentes.
- Novas ferramentas (incluindo novas versões das que já existem) vão aparecer.

Por outro lado,

**Em geral não é possível usar sempre a melhor ferramenta do mundo para cada tipo de problema:**

- Códigos em geral contém problemas de várias categorias diferentes, e misturar linguagens nem sempre é prático.
- A disponibilidade (para o autor e usuários), e o conhecimento prévio do autor podem fazer com que a linguagem mais ideal para uma classe de problema não seja conveniente para o código sendo desenvolvido.

Ex: Perl é a melhor para expressões regulares (regex)\*, mas ninguém vai aprender Perl apenas para um único uso de regex. É mais fácil usar a linguagem já bem conhecida, mesmo que seja um pouco mais desajeitada para regex.

\*Expressões regulares serão discutidas em uma aula futura.

# Opções e escolha de linguagens

**Escolha de linguagem a aprender, ou a usar em um projeto, é em geral um compromisso.**

Se o código começar a ficar muito desajeito ou problemático, pode ser um sinal de que a escolha não está (não está mais) sendo a melhor.

Quem vai trabalhar com problemas computacionais tem grande chance de economizar tempo se gastar tempo para aprender formas melhores de resolver os problemas.

**O conhecimento do autor gera um viés para que seja mais fácil usar a linguagem que conhece melhor, mesmo que outra a princípio seja mais fácil para o problema.**

**Preferência pessoal (de linguagem, estilo, organização, algoritmos, bibliotecas) também é relevante para escolher a melhor solução:** com as escolhas mais confortáveis, pode ser mais rápido e confiável resolver o problema.

Usar uma opção por escolha é diferente de a usar por não conhecer alternativas:

- Familiaridade com as opções disponíveis é importante para fazer escolhas informadas.

# Características de linguagens

Diferentes linguagens foram criadas para diferentes problemas, por diferentes escolhas.

Ao contrário de linguagens humanas, linguagens de programação são criadas de forma sistemática, com objetivos e regras bem definidos. As regras e suas exceções não são arbitrárias, são deliberadas.

**Entender a idéia e as escolhas feitas para uma linguagem ajuda a entender como funcionam e para quê.**

Critérios mais importantes (discutidos individualmente adiante):

1. **“Tipo” (compilada X interpretada, estática X dinâmica)**
2. Popularidade (sim, é relevante)
3. Disponibilidade (custo, plataformas disponíveis, portabilidade)
4. Estruturas de dados implementadas
5. Arrays de mais de 1D (de importância freqüente em ciências computacionais)
6. Conteúdo das bibliotecas (padrão e comuns):
  - 6a. Visualização
  - 6b. Armazenamento de dados
  - 6c. Strings e expressões regulares (regex)
  - 6d. Rotinas científicas (matemática, estatística, processamento de imagens, etc.)
  - 6e. Tarefas gerais (acesso ao sistema, interfaces gráficas, rede, documentação, introspecção, unit tests, compatibilidade com outras linguagens, etc)
7. Outras

**As discussões e comparações adiante vão em geral omitir, mais notadamente: MATLAB, Mathematica, Ruby, Objective C, C#, Lisp, SQL, Visual Basic, e shells.**

# Características de linguagens - “tipos” de linguagens

As duas mais importantes classificações para linguagens:

- Código compilado X código interpretado (não mutuamente exclusivos)
- Tipos estáticos X tipos dinâmicos

São categorias independentes, embora freqüentemente correlacionadas (linguagens interpretadas costumam ter tipos dinâmicos).

Determinam o domínio da maior parte das possibilidades de uma linguagem.

# Características de linguagens – compiladas X interpretadas

Todas as linguagens de uso comum são de relativamente alto nível:

- Funcionam com abstrações próximas do usuário
- Distantes do baixo nível (código de máquina) usado pelos processadores

# Características de linguagens – compiladas X interpretadas

Todas as linguagens de uso comum são de relativamente alto nível:

- Funcionam com abstrações próximas do usuário
- Distantes do baixo nível (código de máquina) usado pelos processadores

Ninguém hoje vai escrever código de máquina:

Adding the registers 1 and 2 and placing the result in register 6 is encoded:

[	op		rs		rt		rd	shamt	funct]		
	0		1		2		6		0	32	decimal
	<b>000000</b>		<b>00001</b>		<b>00010</b>		<b>00110</b>		<b>00000</b>	<b>100000</b>	<b>binary</b>

# Características de linguagens – compiladas X interpretadas

Todas as linguagens de uso comum são de relativamente alto nível:

- Funcionam com abstrações próximas do usuário
- Distantes do baixo nível (código de máquina) usado pelos processadores

Ninguém hoje vai escrever código de máquina:

Adding the registers 1 and 2 and placing the result in register 6 is encoded:

```
[ op | rs | rt | rd |shamt| funct]
   0   1   2   6   0   32   decimal
000000 00001 00010 00110 00000 100000  binary
```



Apollo Guidance Computer, 1964 (76KB, 32 kg)

Interface

Tabela de comandos

```
VERB LIST
50 PLEASE PERFORM
51 PLEASE MARK
52 PLEASE MARK ALT LOG
54 SEND COAS MARK
56 TERMINATE P20
58 STICKFLAG (0) YSONTB FLAGST
64 OPTICS ANGLE TRANSFORM
74 ENABLE VHF DATA PROC
77 DISABLE VHF DATA PROC
85 SEND PARAM DISP NO 2
86 REJECT SEND COAS MARK
87 ENABLE VHF RANGE MARKS
88 DISABLE VHF RANGE MARKS
93 ENABLE W-MATRIX INIT
96 TERM SV INTEG (CALL POOL)

25 PLEASE PERFORM
28 TID NC2
33 TIG
34 T EVENT (PROGRAM)
37 TIG TPI
40 TF GETITEC-VG-AV
45 MARKS-TF GETI-RGA
49 A POS-A VEL-CODE
53 RANGE-RR-PHI
54 RANGE-RR-THETA
56 VEHICLE RATE
59 ATPI-AVTPF-ATZ
59 AVLOS X-AVLOS Y-AVLOS Z
70 SENSOR/COE (BEFORE MK)
71 SENSOR/COE (AFTER MK)
```

```
NOUN LIST
04 ATT ERR
05 ANG SEP ERR-ANG SEP
09 OPTION CODE

72 TIME OF OPT
75 SHNR-ATI-ATZ
76 RANGE-RR-TIME FR OPT
77 RANGE-RR-THETA/PHI
78 YAW-PITCH-ROLL
```



# Características de linguagens – compiladas X interpretadas

Compilador/interpretador traduz de uma linguagem de alto nível, com abstrações (*código fonte*) para a linguagem de máquina (*executável / binário*).

Distinção pelo momento da tradução:

- **Compiladores traduzem previamente**, quando o software está sendo desenvolvido. Na hora do uso só o executável é usado.
- **Interpretadores fazem a tradução “ao vivo”**, na hora do uso do programa.

Interpretação tem um custo possivelmente relevante → código interpretado **pode ser** mais lento que o previamente compilado.

Compilação acontece em um momento em que se tem menos informações do que na execução

- mais difícil gerar código que descubra as condições de execução (exs: tamanhos de arquivos, dimensões de arrays)
- mais difícil haver flexibilidade, introspecção e comportamento dinâmico (mais detalhes adiante)

# Características de linguagens – compiladas X interpretadas

Linguagens não precisam ser exclusivamente compiladas ou interpretadas:

- Algumas dão a opção de compilar ou interpretar: **IDL, Python**
- Algumas são compiladas para um código intermediário (*bytecode*) independente de plataforma, que é executado por uma máquina virtual (VM), que faz uma interpretação ou compilação para o código de máquina nativo: **Java, IDL**
- A definição em geral não é completamente intrínseca à linguagem, e a opção alternativa em princípio pode ser possível, embora possa não ter sido implementada, ou ser muito rara ou inconveniente.
- As interpretadas podem compilar os códigos só uma vez na hora de os executar (não os reinterpretar toda vez). E há a possibilidade de elas os compilarem diretamente para código de máquina nativo, não para bytecode: **Java JIT, Psyco, PyCUDA.**

Em linguagens compiladas, o peso de uma (re)compilação é geralmente grande:

- Pode ser um processo lento e difícil, dependendo de muitos outros recursos externos (compilador, bibliotecas), que o usuário pode nem ter (e nunca chegar a precisar).
- A compilação também pode ter grande efeito sobre a eficiência e a robustez do código.

Programas interpretados têm que ser distribuídos como código-fonte, o que em algumas situações é indesejável – não costuma ser relevante em ciência, mas pode ser importante em casos comerciais.

# Características de linguagens – compiladas X interpretadas

**Linguagens exclusivamente/normalmente compiladas:** C, C++, Fortran

**Linguagens exclusivamente/normalmente interpretadas:** R, Perl, shells

**Linguagens mistas/variáveis:** Java, IDL, Python

# Características de linguagens – compiladas X interpretadas

## Mito 1: *Linguagens compiladas são mais rápidas que interpretadas.*

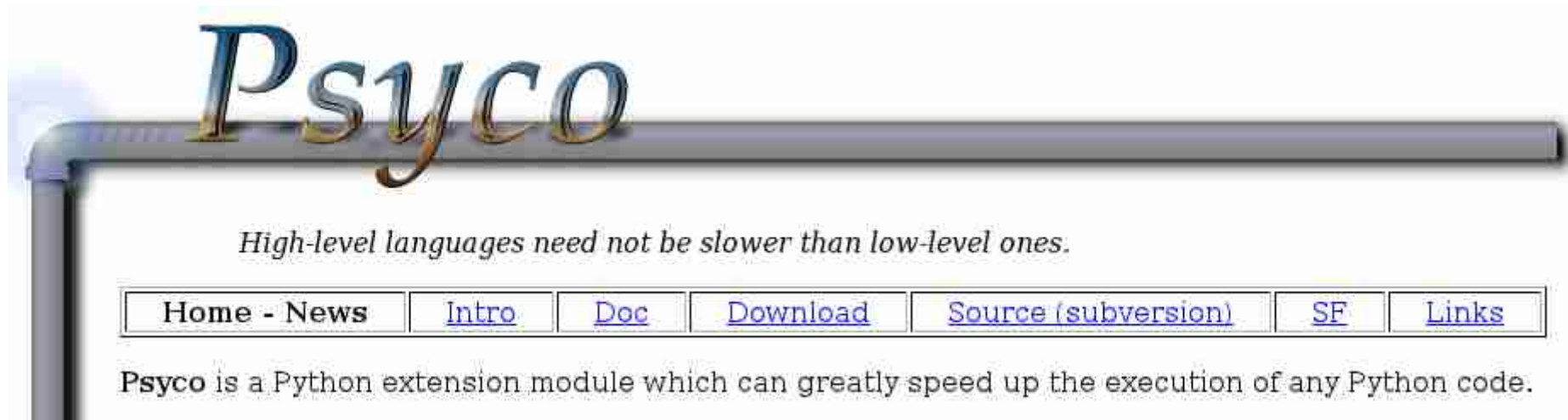
- Interpretação implica trabalho a mais na hora da execução, **mas a diferença pode ser irrelevante.**
- A eficiência de qualquer código (compilado ou interpretado) depende também de como ele foi escrito e/ou compilado.
- O tempo que um código consome não é só tempo de execução: tempo de escrever, consertar, manter e testar o código também é relevante\*
- O tempo perdido para conseguir compilar um código em linguagens compiladas em geral é muito maior que o tempo de compilação na hora de execução em uma interpretada (e muitas vezes é maior que o tempo de execução).

\*muitas vezes, é muito maior que o tempo de execução; por exemplo, a Millenium Simulation, de  $\sim 10^5$  partículas, levou só 28 dias para rodar.

# Características de linguagens – compiladas X interpretadas

## Mito 1: *Linguagens compiladas são mais rápidas que interpretadas.*

- Em muitas interpretadas (IDL, Python, Java), muitas partes mais computacionalmente pesadas são implementadas internamente de forma otimizada.
  - Estas partes (que podem consumir quase todo o tempo) podem já ser impossíveis de serem melhoradas, em qualquer linguagem.
  - A parte restante pode ser computacionalmente leve (não há problema em as fazer em mais alto nível), mas de algoritmo muito complicado (consumiriam muito tempo para implementar, testar e manter em baixo nível).
- As interpretadas podem compilar os códigos só uma vez na hora de os executar (não os reinterpretar toda vez). E há a possibilidade de elas os compilarem diretamente para código de máquina nativo, não para bytecode: **Java JIT, Psyco, PyCUDA:**

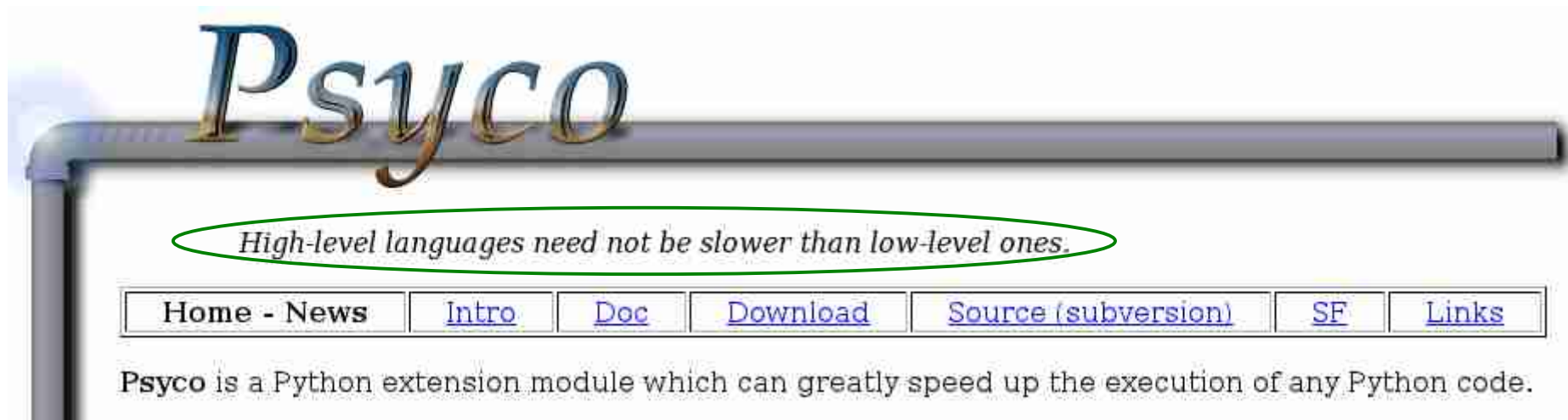


(<http://psyco.sourceforge.net>)

# Características de linguagens – compiladas X interpretadas

## Mito 1: *Linguagens compiladas são mais rápidas que interpretadas.*

- Em muitas interpretadas (IDL, Python, Java), muitas partes mais computacionalmente pesadas são implementadas internamente de forma otimizada.
  - Estas partes (que podem consumir quase todo o tempo) podem já ser impossíveis de serem melhoradas, em qualquer linguagem.
  - A parte restante pode ser computacionalmente leve (não há problema em as fazer em mais alto nível), mas de algoritmo muito complicado (consumiriam muito tempo para implementar, testar e manter em baixo nível).
- As interpretadas podem compilar os códigos só uma vez na hora de os executar (não os reinterpretar toda vez). E há a possibilidade de elas os compilarem diretamente para código de máquina nativo, não para bytecode: **Java JIT, Psyco, PyCUDA:**



(<http://psyco.sourceforge.net>)

# Características de linguagens – compiladas X interpretadas

**Mito 2: *Linguagens compiladas são o mesmo que estáticas, e linguagens interpretadas são o mesmo que dinâmicas.***

- É comum que as compiladas sejam estáticas e as interpretadas sejam dinâmicas, mas isso é só uma coincidência, que nem sempre ocorre.
- Por isso o mito 1 muitas vezes é aplicado à distinção estática/dinâmica.

# Características de linguagens – estáticas X dinâmicas\*

\*tipos estáticos X tipos dinâmicos

## O que é uma variável?

Antigamente, era um só nome para se referir ao conteúdo de um lugar da memória:

No lugar de dizer

*armazene o inteiro 2 na posição 47 (da memória)  
adicione o inteiro 1 ao conteúdo da posição 47  
imprima o conteúdo da posição 47*

Podia-se dizer (em pseudocódigo\*\*)

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

← declaração: um pedido de criação de uma *variável* (uma **instância** do **tipo** `int`) com o nome `number_of_days`

É muito melhor se referir ao nome `number_of_days` do que aos lugares na memória:

- O nome indica o que aquele valor representa
- É mais legível e portátil

Muitos ainda pensam em variáveis apenas assim: só um nome a associar a um número ou string armazenado na memória.

\*\*Nenhuma linguagem específica

# Características de linguagens – estáticas X dinâmicas

## Uma variável é algo muito mais elaborado que só um lugar na memória:

- Uma variável é uma representação de um **tipo** de informação no programa.
- Um **tipo** é uma abstração para um conceito.
  - Exs: inteiros, reais, strings (texto)\*, complexos, etc.
- **Internamente, não existem estes conceitos:** Não há um número real na memória. Há uma representação (binária) dele, através das regras definidas pelo tipo real.
- **Esta abstração inclui regras para seu uso.** Exs:
  - Somar 2 inteiros não é a mesma operação que somar 2 reais (*floats, doubles*). O processador usa regras diferentes para operar sobre valores inteiros ou sobre reais.
  - **3/2** é **1**, enquanto **3.0/2.0** é **1.5**
  - **acos(2.0)** não existe para reais, mas existe para complexos
  - Codificação de strings (**muitas** possibilidades diferentes)\*
  - Regras para ordenar strings (podem variar em critérios como ordem entre números, maiúsculas, minúsculas, etc.)\*

\*Strings serão discutidos só na última aula

# Tipos de variáveis

## Uma variável pode ser muito mais complicada que um número ou string:

- **Contêiner:** armazena vários valores, organizados por ordem, nome, ou hierarquia.
  - Exs: vetor, matriz, array, lista, mapa, árvore, etc.  
(outra aula)
- **Vários valores de tipos mais simples, agrupados - Estrutura** (outra aula)
- **Referência a outra variável** - ponteiro (outra aula)
- **Uma abstração que represente um conceito qualquer - objeto** (outra aula)
  - Compreende dados, recursos, e formas de operar sobre eles
  - É uma variável ativa ("*inteligente*"): não é apenas um repositório estático de dados, é algo capaz de realizar operações.

# Características de linguagens – estáticas

**Linguagens de tipos estáticos:** tipos designados no **momento da compilação**

Existem **declarações** de variáveis

Declaração diz ao compilador para criar uma variável de algum tipo (e dimensões, se array), e associar a ela um nome (símbolo):

C, C++

```
int number_of_days;
string file_name;
double image[1024,768];
float temperatures[];
```

Fortran ≥90

```
integer number_of_days
character*70 file_name
real(selected_real_kind(15,307)) :: image(1024,768)
real(selected_real_kind(6,37)), allocatable:: temperatures()
```

**Na hora da compilação** o compilador cria as instruções para alocar estas variáveis, e uma lista de símbolos, que associa os nomes às variáveis na memória

**A lista de símbolos é estática: estes tipos não podem mudar dentro do programa.**

# Características de linguagens – estáticas

Declaração e definição de **variáveis** não são a mesma coisa (ex. C/C++):

```
int a;      —————▶ Declaração  
int b=2;   —————▶ Declaração e definição
```

Declaração e definição de **tipos** também não são o mesmo: a definição em geral é um código longo, que especifica como o tipo vai funcionar (mais detalhes na aula de objetos).

## Propriedades importantes de linguagens estáticas

- Variáveis precisam ser declaradas.
- Variáveis não podem mudar de tipo dentro do programa.
- Os tipos (e em alguns casos, tamanhos\*) precisam ser conhecidos na hora da compilação.
- As interfaces de rotinas usam tipos (e em alguns casos, tamanhos\*) explicitamente especificados para os argumentos.

\*dimensões de arrays e/ou comprimento de strings, dependendo do caso

# Características de linguagens – estáticas

Exemplos de linguagens estáticas: Fortran, C, C++, Java

Em algumas (C, C++, Java) há escopo de blocos: Variáveis podem ser declaradas dentro de blocos, e só existem ali:

```
for (int n=0; n<10; n++) {
    int a=n;
}
```

**n** e **a** são declarados dentro do loop: só existem dentro do loop; ajuda a manter o código mais limpo e diminui a chance de erros

Em algumas estáticas (Fortran), as declarações têm que estar no começo da unidade (programa/função/subrotina): muito ruim para a organização do código.

Em F77, tipos e dimensões são estáticos: só no F90 apareceu alocação dinâmica de memória (para que não seja necessário saber todas as dimensões na hora da compilação).

Em C sempre houve alocação dinâmica, mas explícita e de baixo nível (**malloc**, **free**).

C++ e Java sempre tiveram tipos com alocação dinâmica (dimensões de arrays e comprimentos de strings podem variar), embora os tipos ainda sejam estáticos.

Em Fortran, o comprimento de strings (tipo **character**) normalmente é estático (tem que ser definido na declaração). Só no F95 apareceu (como módulo opcional) um padrão de strings de comprimento variável, mas até hoje poucos compiladores as suportam.

Tipos implícitos de Fortran ainda são estáticos: são apenas uma convenção de que se a variável não foi declarada, o compilador assume uma declaração default. E seu uso é desaconselhado, por não deixar explícitos os tipos, facilmente levando a erros.

# Características de linguagens – dinâmicas

Linguagens de tipos dinâmicos: tipos designados no momento da execução da linha de código onde acontece uma atribuição (a um nome não qualificado).

Exemplos: **IDL**, **Python**, **R**, **Perl**.

As variáveis contém, adicionalmente, **metadados**, para obter informações sobre as características da variável (tipo, tamanho) no momento da execução, já que em geral estas não eram conhecidas quando o código foi escrito (não podem ser assumidas), e nem quando foi compilado (se foi compilado).

# Características de linguagens – dinâmicas

Obter informações sobre variáveis é uma das formas de *introspecção*, algo essencial em linguagens dinâmicas:

```
IDL> help,some_variable
SOME_VARIABLE UNDEFINED = <Undefined>      some_variable ainda não existe
IDL> some_variable=2
IDL> help,some_variable
SOME_VARIABLE INT = 2                        Agora, é um int
IDL> some_variable=[2d0,5d0,74.327d0,!dpi]
IDL> help,some_variable
SOME_VARIABLE DOUBLE = Array[4]            Agora, é um array de 4 doubles
IDL> print,some_variable
      2.00000000      5.00000000      74.327000      3.1415927
IDL> print,n_elements(some_variable)
      4
IDL> some_variable=!null
IDL> help,some_variable
SOME_VARIABLE UNDEFINED = !NULL
IDL> print,n_elements(some_variable)
      0
Agora, é não definida de novo
```

# Características de linguagens – dinâmicas

Em linguagens de tipos dinâmicos não existe declaração de variáveis.

Ex (IDL):

```
temperature=dblarr(3)
```

Não está declarando uma variável. `dblarr(n)` é uma função, que retorna um array de `n` elementos. À variável `temperature` está sendo atribuído o resultado desta função.

Em

```
days=intarr(3)  
days=[9,5,7]
```

A primeira linha não deveria existir: a segunda está primeiro apagando o que existia com o nome `days`, e então criando uma variável `days` com o conteúdo `[9,5,7]`: **o array criado na primeira linha é jogado fora na segunda linha, ele nunca é usado.**

# Características de linguagens – dinâmicas

*Introspecção* sobre variáveis permite que em linguagens dinâmicas todas as rotinas sejam genéricas com relação ao tipo e dimensões de seus argumentos.

O que simplifica muito escrever interfaces:

Em uma linguagem estática, todos os argumentos têm que ser declarados de forma idêntica na rotina os recebe, e onde aquela rotina é chamada.

Ex (Fortran):

```
function average(num1,num2)
implicit none
real(selected_real_kind(15,307)), intent(in) :: num1, num2
real(selected_real_kind(15,307)) :: average
average=(num1+num2)*0.5d0
end function average
```

```
program use_average
implicit none
real(selected_real_kind(15,307)) :: num1=3d0, num2=4d0
real(selected_real_kind(15,307)) :: average
print '(E15.5)', average(num1,num2)
end program use_average
```

Seria um erro ter usado

```
print '(E15.5)', average(3.0, 4)
```

Porque os argumentos são do tipo errado (um é precisão simples, o outro é inteiro).

# Características de linguagens – dinâmicas

Em uma linguagem dinâmica, a função não especifica os tipos dos argumentos:

O mesmo exemplo em IDL:

```
function average, num1, num2  
return, (num1+num2)*0.5d0  
end
```

```
pro use_average  
num1=3d0  
num2=4d0  
print, format='(E15.5)', average(num1, num2)  
end
```

Seria válido ter usado

```
print format='(E15.5)', average(3.0, 4)
```

E até com arrays:

```
print, format='(E15.5)', average([3d0, 9d0, 11d0], 4d0)
```

o que teria resultado em

```
3.50000E+00  
6.50000E+00  
7.50000E+00
```

# Características de linguagens – estáticas X dinâmicas

Tipos dinâmicos simplificam muito a programação, especialmente em interfaces de rotinas.

Em linguagens dinâmicas:

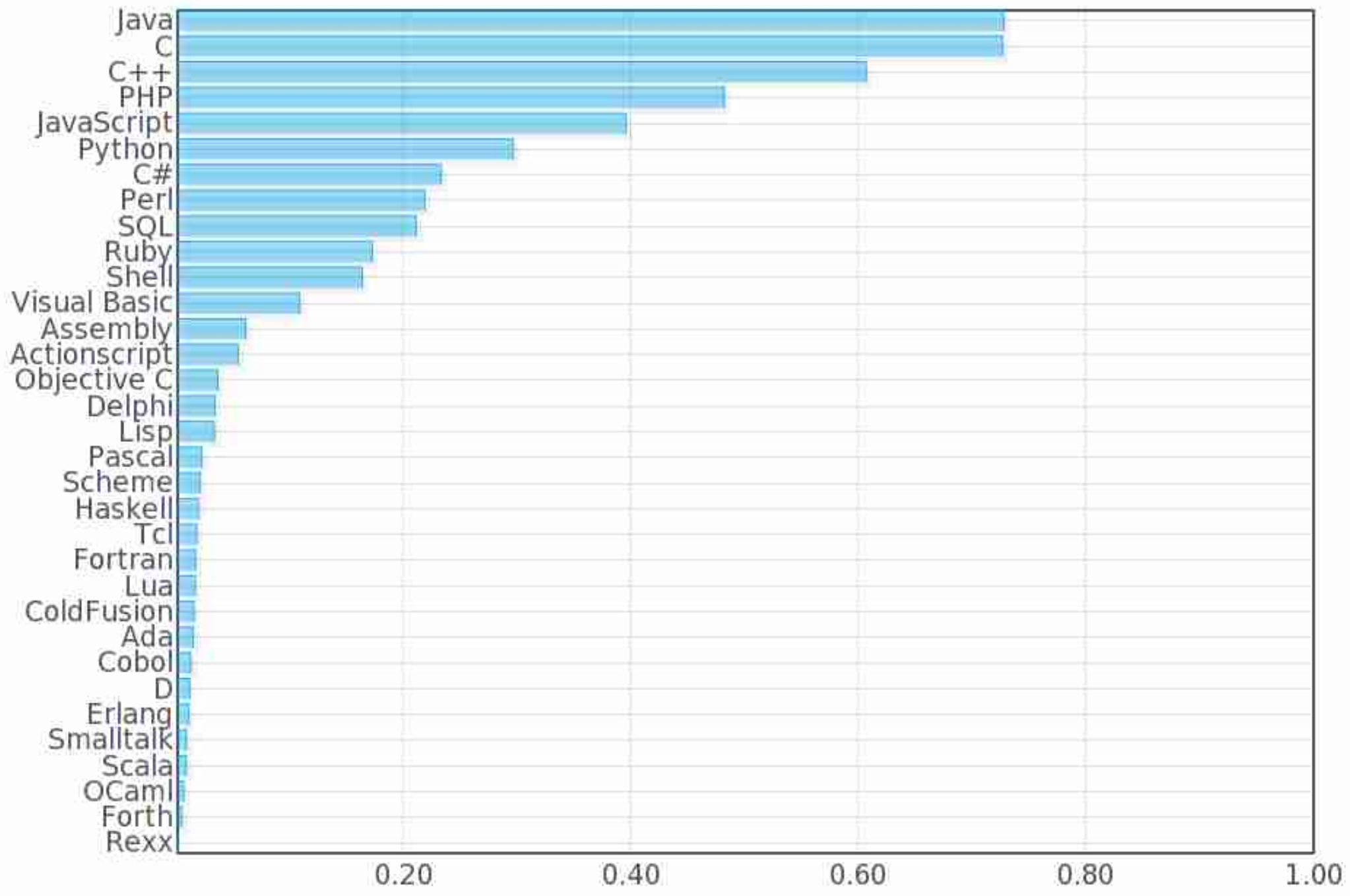
- É simples escrever interfaces de rotinas (inclusive pelo uso comum de argumentos nomeados (*keywords*), não só posicionais).
  - O que ajuda a manter o código mais estruturado, e portanto melhor organizado e mais compreensível e reusável.
- Mas como o compilador/interpretador não sabe os tipos/tamanhos na hora da compilação, ele não tem como verificar erros
  - Ele nem tem como saber o que é um erro; é algo que só o programador sabe.
- Se uma rotina precisa assumir tipos/tamanhos específicos para seus argumentos, ela deve os verificar, por introspecção, na hora da execução.
  - O que em geral não é uma desvantagem.
- Devido aos metadados que fazem parte das variáveis, escalares ocupam mais memória, e são de acesso mais lento que em linguagens estáticas
  - Nas estáticas a contabilidade e verificações foram feitas pelo compilador, não acontece na hora da execução.
  - Irrelevante para arrays, onde só há um conjunto de metadados para todos os muitos elementos do array.

# Características de linguagens – popularidade

Sim, é um ponto relevante. Para as linguagens dominantes:

- Há (boas e bem testadas) rotinas prontas para quase toda tarefa geral, e para muitas das tarefas comuns nas áreas em que ela são usadas.
- Há muitas referências para as aprender.
- Há muitos usuários para fornecer ajuda (e muitos que já fizeram as mesmas perguntas).
- Há muitos usuários encontrando os bugs, muitos dos quais já vão ter sido consertados.
- É mais fácil compartilhar código.
- É mais fácil que elas estejam disponíveis / sejam instaladas.

# Características de linguagens – popularidade (langpop.com)



As linguagens científicas (IDL, R, MATLAB, Mathematica) nem aparecem na lista.

# Características de linguagens – popularidade

Programação científica deixou de ser uma grande fração da programação feita no mundo.

A maior parte das linguagens/recursos em programação, hoje, se referem a programação geral, que não encontra muitos dos problemas específicos de programação científica.

O uso das linguagens tem o viés de o quão comuns são as áreas a que elas se destinam:

• **IDL** é predominantemente usado em astronomia, “ciências da terra” (geofísica, sensoriamento remoto, meteorologia) e imageamento médico; **é desconhecido fora destas áreas.**

- Apesar de ser obscuro, IDL tem a vantagem de ter um newsgroup ativo, lido pelos principais especialistas e desenvolvedores, que coletivamente conhecem todos os detalhes de IDL.
- Outras linguagens, mais populares, não têm um fórum tão centralizado (o especialista de uma área específica pode não ler aquele mesmo fórum), ou têm um volume muito grande de tráfego (o que dificulta que uma pergunta seja notada ou receba muita atenção).
- Apenas Python+NumPy é comparável a IDL em arrays de mais de 1D. Nenhum outro ambiente citado aqui é comparável a IDL em visualização.

# Características de linguagens – popularidade

- **R é a única linguagem para Estatística** (estatística não é só desvio padrão, Gaussianas e mínimos quadrados).
  - Não por propriedades intrínsecas da linguagem; se deve apenas à quantidade de rotinas estatísticas implementadas em R (IDL e Python são linguagens intrinsecamente muito melhores para implementar rotinas estatísticas, mas ninguém o fez, e ninguém vai querer fazer, dado que teria que implementar do nada).
- MATLAB parece ser mais comum em engenharia.
- Mathematica parece ser mais comum em matemática e física teórica.
- Python+NumPy+SciPy+Matplotlib (têm ~5 anos de vida) tem crescido, tomando lugar destas acima. Provavelmente em parte por Python ser uma linguagem **moderna, gratuita, e de muita capacidade para uso geral**.
  - Mas não é tão portátil como Java ou IDL. Especialmente agora, enquanto não termina a transição para Python 3 (só em 2010 surgiu NumPy para Python 3).

# Características de linguagens – disponibilidade

## Portabilidade

- Em poucas linguagens o código pode ser usado, inalterado, em várias plataformas.
- As interpretadas em geral são mais portáveis, pois o código é executado por um interpretador / máquina virtual, não diretamente em cada sistema.
- **Java é a mais portátil**, com o código sendo executado pela máquina virtual (VM) que é uniforme. A VM de Java é normalmente gratuita e já está em todo computador.
- IDL, R e Python, em graus menores podem ser também.
- C, C++ e Fortran são as menos portáveis: é comum dependerem de bibliotecas externas de variável disponibilidade, dependerem de particularidades do sistema, e o mesmo código pode nem funcionar em um compilador diferente no mesmo computador, ou na mesma plataforma em computadores diferentes (por diferenças pequenas no software).

## Custo

- Java, Python, R, Perl normalmente são gratuitas.
- IDL normalmente tem custo (é possível, com limitações, gerar distribuições de rotinas IDL que podem ser executadas sem licença). GDL e FL são abertas, mas muito limitadas.
- C, C++, Fortran normalmente têm opções gratuitas, mas vários dos principais compiladores não são gratuitos – **e nem todo código funciona em todo compilador (principalmente Fortran).**

# Características de linguagens – disponibilidade

## Disponibilidade

- As linguagens mais obscuras (IDL, Python 3, R, Fortran 2003/2008) são mais difíceis de encontrar em variadas plataformas, principalmente nas que não são as 3 padrão (Windows, Linux, Mac).
- Normalmente, “qualquer coisa” (até celulares) tem uma máquina virtual Java e compilador de C. Mas só Java é uniforme; C tem muitas dependências de plataforma (principalmente bibliotecas).

# Características de linguagens – estruturas de dados\*

Estruturas disponíveis (principalmente na biblioteca padrão) são um dos pontos mais importantes para determinar que tipo de código

- Em especial, quão complexos os algoritmos) pode ser implementado de forma razoável.

Linguagens dinâmicas naturalmente têm uma grande vantagem em flexibilidade de estruturas de dados (principalmente as dinâmicas e não homogêneas).

- Obviamente, strings e arrays têm alocação dinâmica.

\*A importância destas estruturas será discutida em outra aula. Ponteiros não são estruturas de dados, mas são importantes para elas.

# Características de linguagens – estruturas de dados\*

Começando das mais simples (cada nível inclui o que têm os níveis mais baixos):

1 - Fortran 77 (só arrays estáticos (tamanho declarado no código-fonte)).

2 - C - na biblioteca padrão, só arrays, estruturas, e ponteiros, embora haja grande variedade em outras bibliotecas.

3 - Fortran 90 (só arrays e estruturas, com muitas limitações, e ponteiros primitivos).

4 - Fortran 2003/2008 - menos limitado que F90, com objetos rudimentares.

4 - R, Perl – arrays e strings, algumas opções em listas heterogêneas, objetos simples.

5\*\* - C++, Java - grande variedade na biblioteca padrão, incluindo várias implementações de strings e arrays dinâmicos, estruturas, ponteiros, contêineres (listas, mapas, árvores, conjuntos) homogêneos, e objetos muito bem desenvolvidos.

5\*\* - IDL 8, Python+NumPy - arrays multidimensionais (excepcionalmente bem), estruturas, listas e mapas heterogêneos, objetos bem desenvolvidos. Em todos estes aspectos Python+NumPy é melhor desenvolvido que IDL.

\*\* Quais dos listados nos (5) são mais desenvolvidos depende das estruturas sendo consideradas.

# Características de linguagens – arrays multidimensionais (MD)

Arrays com mais de 1D têm importância predominantemente em computação científica.

- Por isso, quase não recebem atenção em linguagens para programação geral.
- 2D e 3D têm alguns usos em computação gráfica, mas com frequência são simples.

A importância e usos de arrays MD será discutida em outra aula.

Sem bom suporte semântico a operações MD, torna-se extremamente desajeitado operar em muitos dos dados que ocorrem em computação científica.

A maior parte dos programadores científicos não faz uso destas possibilidades, o que gera muito sofrimento desnecessário e código muito complicado e frágil.

# Características de linguagens – arrays multidimensionais (MD)

Começando das mais avançadas:

- 1. Apenas IDL (desde 1977) e Python+NumPy (desde 2005) têm bom suporte a arrays MD. Python+Numpy é o mais poderoso (mas também mais complexo\*).**
  - **As outras linguagens são desprezíveis em comparação.**
2. R - Menos limitado que os outros abaixo.
3. Fortran 90/95/2008 – apesar das limitações (principalmente na falta de objetos) e semântica e biblioteca padrão limitadas, não é tão desajeitado para mais de 1D (mas é muito pior que C++ e Java para 1D). As limitações diminuem um pouco do 90 ao 2008.
4. Java, C++ - as bibliotecas padrão têm muito bom suporte a 1D, mas são desajeitadas para mais de 1D. A biblioteca boost (não padrão) ajuda em C++.
5. C - semântica simples, sem operações vetoriais, e alocação dinâmica trabalhosa.
6. Fortran 77, Perl - impraticáveis, por falta de alocação dinâmica (F77), falta de operações vetoriais (F77), e falta de MD de verdade (Perl).

\*Lei de Parker: *Com grande poder (às vezes) vem grande complexidade.*

# Características de linguagens – conteúdo das bibliotecas

**A escolha de linguagens não depende só das características da linguagem em si.**

A sua biblioteca padrão (que a acompanha, sempre disponível) é de grande importância para determinar o que se pode fazer facilmente com ela.

Bibliotecas não padrão têm importância menor, porque podem não estar disponíveis onde o código vai ser usado, mas são também de grande relevância.

- Em particular são relevantes as “quase padrão”, como **NumPy**, **SciPy** e **Matplotlib** (Python), e **idlastro** e **Coyote** (IDL).

Nas linguagens dinâmicas, as bibliotecas padrão costumam ter, adicionalmente,

- Introspecção - para descobrir as condições em que o programa está sendo executado  
→ Exs: Existência e propriedades de variáveis, saber como foi chamada a rotina.
- Metaprogramação - avaliação dinâmica de código fonte (ex: executar um trecho de código especificado em uma string), e decisões sobre como executar/compilar rotinas.

# Características de linguagens – conteúdo das bibliotecas

## Conteúdos das bibliotecas padrão (linguagens estáticas):

- Fortran - muito simples, quase que exclusivamente apenas funções matemáticas, operações simples em strings, e acesso rudimentar a entrada/saída e ao sistema.
- C - de baixo nível, mas de escopo relativamente grande para programação geral: matemática, strings (em vários sistemas de caracteres), acesso detalhado ao sistema, entrada/saída mais complexas.
- C++ - inclui tudo de C, mais: (muitos, e elaborados) contêiners, muito mais elaborada para strings (inclusive regex), acesso ao sistema, gerenciamento de exceções, entrada/saída, sistemas de caracteres, e utilidades em geral.
- Java - semelhante em abrangência a C++, mas de forma menos complexa, e com, adicionalmente: interfaces gráficas, rede, segurança, documentação e mais entrada/saída.
- R - simples para tarefas gerais (strings (com regex), sistema, entrada/saída), simples para vetorização e contêiners.
  - **A única extensiva em Estatística.**
  - De boa abrangência para matemática e processamento de dados.
  - Visualização de alto nível, mas simples comparada a IDL.

# Características de linguagens – conteúdo das bibliotecas

- IDL
  - Excepcionais arrays MD (só Python+Numpy está no mesmo nível).
  - Muito boa introspecção e metaprogramação.
  - Suporte a vários formatos de arquivos.
  - De boa abrangência para matemática e processamento de dados.
  - Limitada para interfaces gráficas e operação com outras linguagens.
  - A partir do 8, listas e hashes heterogêneos.
  - **Excelente para visualização.**
  - **Limitada em tarefas mais gerais, como acesso ao sistema, rede, banco de dados e documentação.**
- Python+NumPy+SciPy+Matplotlib
  - Semelhante em abrangência e características a IDL.
  - Visualização mais rudimentar que IDL.
  - **Muito mais suporte a tarefas gerais** (abrangência semelhante à do Java (sem segurança)), com melhor suporte a bancos de dados e unit tests.
  - Excelente em introspecção e metaprogramação.
- Perl - **o padrão em expressões regulares e processamento de strings.**
  - Conteúdos científicos inexistentes ou muito limitados.
  - Conteúdos gerais mais simplificados e limitados que linguagens como Python ou Java.

# Características de linguagens – complexidade

**Complexidade da linguagem não é o mesmo que complexidade do código escrito nela.**

**Linguagens mais simples exigem códigos mais complicados.**

- Simplicidade costuma vir às custas de a linguagem ser mais ingênua, menos completa, de códigos mais longos.

Linguagens simples em geral são mais fáceis de aprender, e é útil conhecer uma linguagem por completo.

- Mas nas complexas em geral é mais fácil e rápido escrever os programas.

As dinâmicas em geral são mais simples de usar, algumas vezes não sendo diretamente comparáveis às estáticas neste ponto.

# Características de linguagens – complexidade

Em ordem crescente de complexidade:

1. Fortran 77 (mais simples)
2. C
3. Fortran 90-2008
4. R
5. IDL
6. Python
7. Java
8. C++ (muito mais complexa que qualquer outra coisa)

Perl é um caso à parte, por ser muito não usual (deliberadamente mais semelhante a linguagens humanas, e não enxuta). Dependendo de como é avaliada, poderia ser de simples/intermediária (~4), a muito difícil (~8).

# Características de linguagens – sistema de objetos

Objetos (discutidos em outra aula) são importantes para a estruturação, compreensibilidade, e reaproveitamento do código.

Linguagens variam muito no nível de suporte que têm a objetos. Onde eles são mais rudimentares, os objetos são quase inúteis.

Começando da mais simples:

1. Fortran 2003/2008

2. R\*

3. Perl

5. IDL

6. Python

7. Java

8. C++

\* R é muito baseada em uso de objetos, mas o sistema para escrever novas classes é rudimentar.

# Características de linguagens

*1940s - Various "computers" are "programmed" using direct wiring and switches. Engineers do this in order to avoid the tabs vs spaces debate.*

*1957 - John Backus and IBM create FORTRAN. There's nothing funny about IBM or FORTRAN. It is a syntax error to write FORTRAN while not wearing a blue tie.*

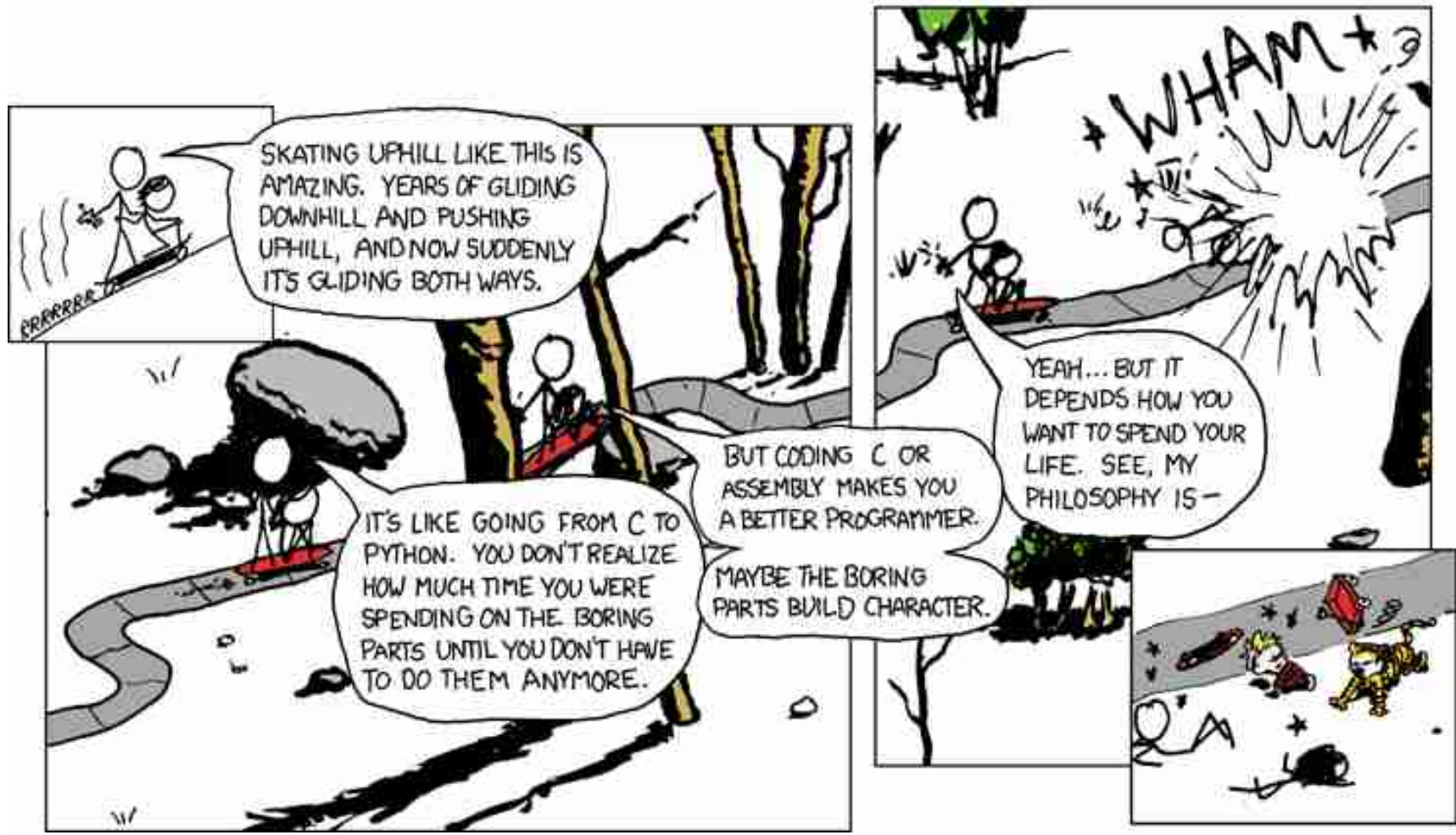
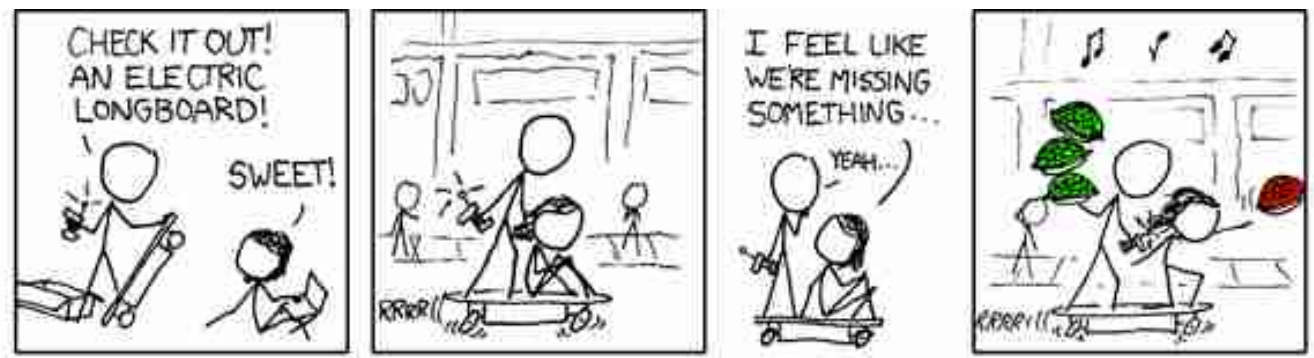
*1972 - Dennis Ritchie invents a powerful gun that shoots both forward and backward simultaneously. Not satisfied with the number of deaths and permanent maimings from that invention he invents C and Unix.*

*1983 - Bjarne Stroustrup bolts everything he's ever heard of onto C to create C++. The resulting language is so complex that programs must be sent to the future to be compiled by the Skynet artificial intelligence. Build times suffer.*

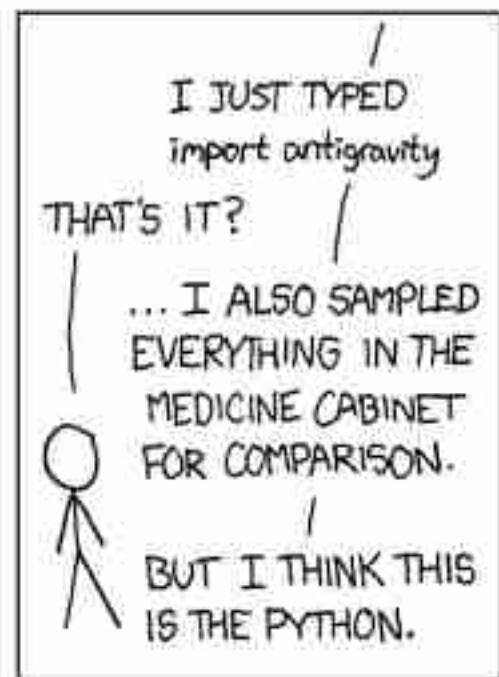
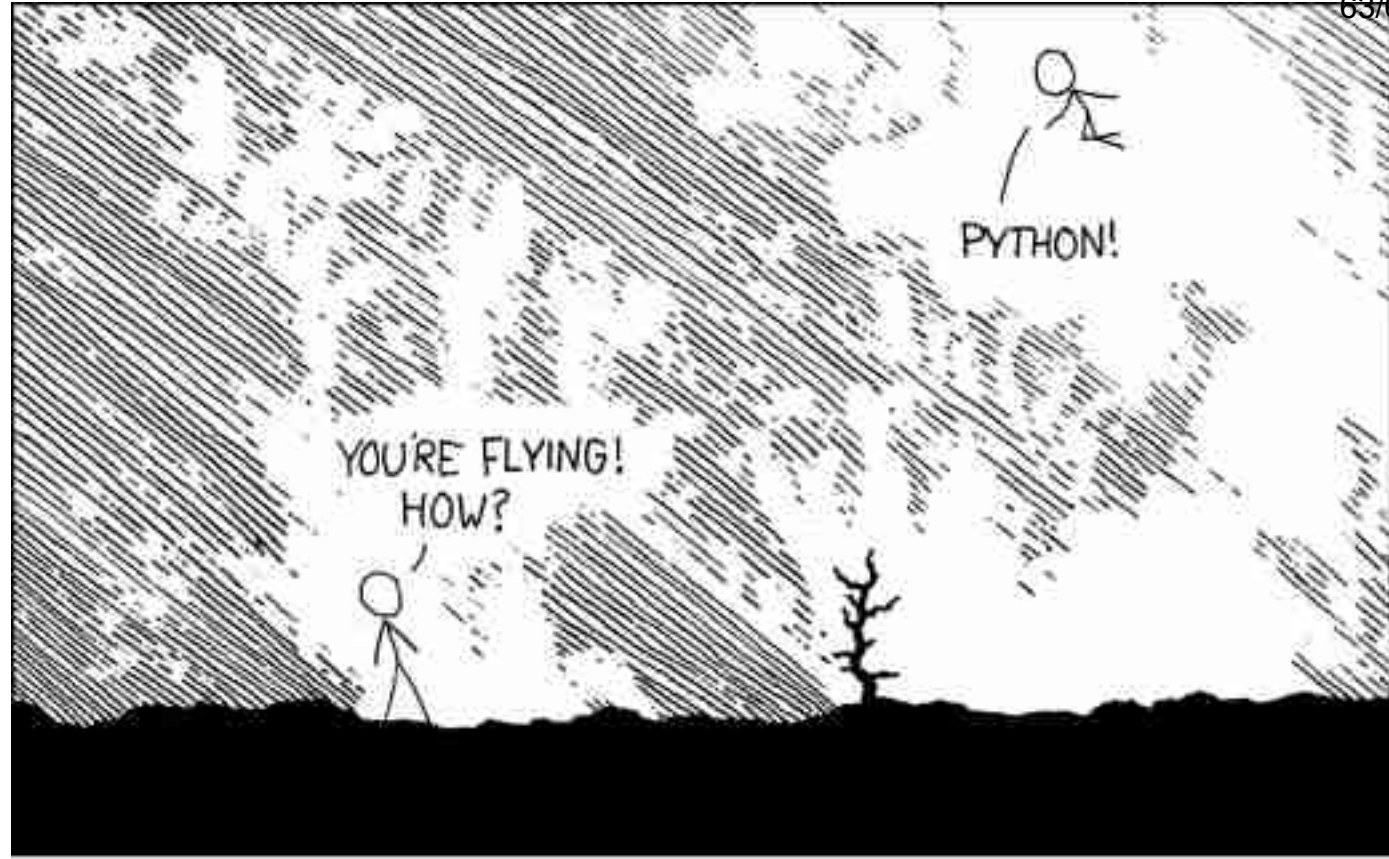
*1987 - Larry Wall falls asleep and hits Larry Wall's forehead on the keyboard. Upon waking Larry Wall decides that the string of characters on Larry Wall's monitor isn't random but an example program in a programming language that God wants His prophet, Larry Wall, to design. Perl is born.*

*De A Brief, Incomplete, and Mostly Wrong History of Programming Languages,  
<http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html>*

# Características de linguagens



# Características de lenguagens



# Bibliotecas

Não faz sentido tentar escrever o código do zero, sem antes procurar (boas) fontes onde o problema possa já ter sido resolvido.

Se o problema é comum (tarefas gerais, formatos de arquivos, operações matemáticas, etc.), provavelmente já existe algo implementado que pode ser aproveitado.

O provido pelas bibliotecas padrão e as mais comuns das linguagens tende a ser o mais robusto (já foi mais usado e testado).

Rotinas individuais, mais obscuras, têm chance maior de ter bugs e de fazer a coisa errada.

Os slides anteriores mostraram as áreas principais de conteúdo das bibliotecas padrão. Outras (não padrão) comuns:

- Python - **NumPy**, **SciPy**, Matplotlib - essenciais para quase qualquer problema científico.
- IDL - **idlastro** (astronomia e utilidades gerais), **IDLDoc** (documentação), **mgunit** (unit testing), **Coyote** (utilidades gerais), **Catalyst** (interfaces gráficas), **GPULib** (CUDA).
- R e Perl têm acervos muito grandes de bibliotecas, em geral relativamente bem testadas e documentadas: **CRAN** e **CPAN** (Comprehensive R|Perl Archive Network).
- Python, Java, C++, C são muito extensivamente usadas, então há muitas bibliotecas (livres e proprietárias), com muita variedade de aplicações.

# Referências (apenas algumas principais)

O quão recente é a referência costuma ser muito relevante.

*Modern IDL*, de Michael Galloy (2011)

*Object-oriented programming with IDL*, de Ronn Kling (2010)

*IDL Primer*, de Ronn Kling (2007)

*Coyote's Guide to IDL Programming*

*Coyote's Guide to Traditional Idl Graphics*, de David Fanning (2011)

<http://www.dfanning.com/> (algumas práticas são antiquadas)

<http://michaelgalloy.com/>

<http://groups.google.com/group/comp.lang.idl-pvwave/topics>

*R in a Nutshell*, de Joseph Adler (2010)

*Java in a Nutshell*, de David Flanagan (2005)

*Programming in Python 3*, de Mark Summerfield (2008)

*Python Scripting for Computational Science*, de Hans Petter Langtangen (2010)

*Matplotlib for Python Developers*, de Sandro Tosi (2009)

*The C++ Programming Language*, de **Bjarne Stroustrup** (1997)

# Referências (apenas algumas principais)

*C Programming Language*, de W. Kernighan e **D. Ritchie** (1988)

*Fortran 95/2003 Explained*, de Metcalf, Reid e Cohen (2004)

*Introduction to Programming with Fortran: with coverage of Fortran 90, 95, 2003 and 77*, de I. D. Chivers (2008)

*Programming Perl*, de **Larry Wall**, T. Christiansen, J. Orwant (2000)

*Software Carpentry*

*Helping scientists make better software since 1997*

<http://software-carpentry.org/>

*Comprehensive Perl Archive Network (CPAN)*

<http://www.cpan.org/>

*Comprehensive R Archive Network (CRAN)*

<http://www.r-project.org/>

Boost C++ libraries

*“provides free peer-reviewed portable C++ source libraries.”*

<http://www.boost.org/>

Safari Books

Muitos livros sobre programação online (incluindo alguns citados acima). Podem ser comprados individualmente, por assinatura, ou **acessados pela assinatura da FAPESP**

<http://proquestcombo.safaribooksonline.com/>

# Sumário

1 – Slides em [http://www.ppenteado.net/pea/pea01\\_linguagens.pdf](http://www.ppenteado.net/pea/pea01_linguagens.pdf)

- Motivação
- Tópicos abordados
- Tópicos omitidos
- Opções e escolha de linguagens
- Uso de bibliotecas
- Referências

“Exercícios” (sugestões de em que pensar após esta aula):

- Que linguagens você usa?
- Por que motivo?
- Para quê?
- Estas linguagens estão atendendo bem às suas necessidades?
- Há código que está ficando muito desajeitado / complicado?
  - Há coisas simples de pensar / descrever, mas cujo código ficou muito complicado? Este é um sinal de que há ferramentas melhores, de mais alto nível (mais próximo às idéias simples de descrever).
- Quanto tempo você gastou estudando as linguagens que usa? Como se compara ao tempo gasto as usando?
- Já procurou saber há ferramentas melhores para as suas necessidades? Já procurou saber se não apareceram coisas melhores, desde a última busca, ou se há alguém trabalhando no que ainda não existe e você precisa?

# Próxima aula

2 – Slides em [http://www.ppenteado.net/pea/pea01\\_organizacao.pdf](http://www.ppenteado.net/pea/pea01_organizacao.pdf)

- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing

Questões que serão discutidas, em que você pode começar a pensar:

- Quão compreensível é para você o código que você usa? (seu ou de outros)
- É fácil saber que código foi usado para produzir que resultados?
- É fácil você reusar o código, especialmente depois de muito tempo? E para outros?
- Qual é a diferença entre o que o código faz e como ele o faz? Ela é clara para quem o vê?
- Quão difícil é resolver problemas (debug)?
- O código foi bem testado? Quão confiável ele é?

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