Universidade Federal de Goiás - UFG

Instituto de Matemática e Estatística

Programa de Mestrado Profissional em Matemática em Rede Nacional

Aritmética com Python

Rogério da Silva Cavalcante

Goiânia – GO

2018

sistema de bibliotecas ufg

PRPG
PRÓ-REITORIA DE PÓS-GRADUAÇÃO

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[X] Dissertação** **[ ] Tese**

2. Identificação da Tese ou Dissertação:

Nome completo do autor: Rogério da Silva Cavalcante

Título do trabalho: Aritmética com Python

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Rogério da Silva Cavalcante
Assinatura do(a) autor(a)[2]

Ciente e de acordo:

Thaynara Arielly de Lima
Assinatura do(a) orientador(a)[2] Data: 01/11/2018

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.
Casos de embargo:
- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

[2] A assinatura deve ser escaneada.

Rogério da Silva Cavalcante

Aritmética com Python

Dissertação apresentada ao Programa de Mestrado Profissional em Matemática em Rede Nacional do Instituto de Matemática e Estatística da Universidade Federal de Goiás, como requisito parcial para a obtenção do título de Mestre em Matemática.

Orientadora: Dra. Thaynara Arielly de Lima

Goiânia – GO
2018

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Cavalcante, Rogério da Silva
Aritmética com Python [manuscrito] / Rogério da Silva Cavalcante. - 2018.
59 f.

Orientador: Profa. Dra. Thaynara Arielly de Lima.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Matemática e Estatística (IME), PROFMAT - Programa de Pós graduação em Matemática em Rede Nacional - Sociedade Brasileira de Matemática (RG), Goiânia, 2018.
Bibliografia.
Inclui tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Lógica. 2. Programação. 3. Aritmética. 4. Python. 5. Ensino de Matemática. I. Lima, Thaynara Arielly de, orient. II. Título.

CDU 51

Universidade Federal de Goiás - UFG
Instituto de Matemática e Estatística - IME
Mestrado Profissional em Matemática em Rede Nacional – PROFMAT/UFG
Campus Samambaia – Caixa Postal 131 – CEP: 74.001-970 – Goiânia-GO.
Fones: (62) 3521-1208 e 3521-1137 www.ime.ufg.br

Ata da reunião da banca examinadora da defesa de Trabalho de Conclusão de Curso do aluno Rogério da Silva Cavalcante – Aos onze dias do mês de outubro do ano de dois mil e dezoito, às 11:00 horas, reuniram-se os componentes da Banca Examinadora: Profª. Drª. Thaynara Arielly de Lima – Orientadora, Prof. Dr. Ole Peter Smith e a Profª. Drª. Eunice Cândida Pereira Rodrigues, para, sob a presidência da primeira, e em sessão pública realizada na sala do LEMAT do IME, procederem a avaliação da defesa intitulada **"Aritmética com Python",** em nível de mestrado, área de concentração Matemática do Ensino Básico, de autoria de Rogério da Silva Cavalcante, discente do Programa de Mestrado Profissional em Matemática em Rede Nacional - PROFMAT da Universidade Federal de Goiás. A sessão foi aberta pela presidente da banca, Profª. Drª. Thaynara Arielly de Lima, que fez a apresentação formal dos membros da banca. A seguir, a palavra foi concedida ao autor do TCC que, em 30 minutos, procedeu à apresentação de seu trabalho. Terminada a apresentação, cada membro da banca arguiu o examinando, tendo-se adotado o sistema de diálogo sequencial. Terminada a fase de arguição, procedeu-se à avaliação da defesa. Tendo em vista o que consta na Resolução nº. 1403/2016 do Conselho de Ensino, Pesquisa, Extensão e Cultura (CEPEC), que regulamenta os Programas de Pós-Graduação da UFG, e procedidas as correções recomendadas, o Trabalho foi **APROVADO** por unanimidade, considerando-se integralmente cumprido este requisito para fins de obtenção do título de **MESTRE EM MATEMÁTICA**, na área de concentração Matemática do Ensino Básico pela Universidade Federal de Goiás. A conclusão do curso dar-se-á quando da entrega, na secretaria do IME, da versão definitiva do trabalho, com as devidas correções supervisionadas e aprovadas pelo orientador. Cumpridas as formalidades de pauta, às 13:00 horas, a presidência da mesa encerrou a sessão e, para constar, eu, Sóstenes Soares Gomes, secretário do PROFMAT/UFG, lavrei a presente ata que, após lida e aprovada, segue assinada pelos membros da Banca Examinadora em duas vias de igual teor.

Thaynara Arielly de Lima

Profª. Drª. Thaynara Arielly de Lima
Presidente – IME/UFG

Prof. Dr. Ole Peter Smitih
Membro – IME/UFG

Profª. Drª. Eunice Cândida Pereira Rodrigues
Membro - UFMT

Rogério da Silva Cavalcante graduou-se em Matemática pela Universidade Federal de Goiás no ano de 2006. Atua como professor do ensino básico desde 2000, já lecionou em escola estaduais, municipais e desde 2013 está no IFG Câmpus Itumbiara.

Dedico este trabalho aos meus pais,
Marluce e Jeremias pela confiança e respeito.

Agradecimentos

Agradeço a Deus por tudo que tem me proporcionado.

Aos meus pais Jeremias e Marluce pelo modo ímpar com que me educaram e por todo o suporte moral, psicológico e financeiro sem o qual essa conquista não se concretizaria.

Agradeço à minha companheira Edilene Cristina que nesse árduo período de lutas e renuncias, demonstrou fé e esperança na conclusão exitosa de tal jornada.

As minhas irmãs Andréa, Alean e Ariadna e ao meu irmão Renato pela torcida e apoio.

Um agradecimento especial a professora doutora Thaynara Arielly de Lima pela impecável orientação, constante incentivo e contínua compreensão que culminaram na realização deste trabalho.

Aos colegas da Escola Municipal Honestino Monteiro Guimarães, em destaque para Talita, Jeovane e Gilson cujas saudosas conversas me impulsionavam para um aprimoramento pessoal e profissional.

Aos colegas do IFG câmpus Itumbiara pelo apoio, diálogos e notória ênfase na importância da conclusão de tal etapa, em especial para Leonardo Garcia Marques pela disposição em compartilhar tempo e conhecimentos sobre LaTeX, programação e Python.

Por fim agradeço a CAPES, SBM, IMPA e UFG por ofertarem novas possibilidades através do PROFMAT.

DON'T PANIC.
(Douglas Adams)

Resumo

Objetivamos neste trabalho mostrar o quão eficazes podem ser os recursos tecnológicos quando utilizados para o ensino. Escolhemos utilizar a linguagem de programação Python no ensino de matemática, especificamente aritmética, exibindo como a lógica transita naturalmente entre a computação e a matemática, mostrando desta maneira uma interdisciplinariedade nata entre esses assuntos. São propostos programas construídos a partir das demonstrações de alguns resultados clássicos em aritmética, visando integrar o ensino desses resultados ao ensino de ferramentas básicas de uma linguagem de programação.

Palavras-chave: Lógica. Programação. Aritmética. Python. Ensino de Matemática.

Abstract

We aim to show how effective the technological resources can be when used for teaching. We chose to use the Python programming language in mathematics teaching, specifically arithmetic, showing how logic moves naturally between computing and mathematics, thereby showing a natural interdisciplinarity between these subjects. We propose programs built from the demonstrations of some classic results in arithmetic, aiming to integrate the teaching of these results to the teaching of basic tools of a programming language.

Keywords: Logic, Programming, Python, Arithmetic, Teaching Mathematics.

Lista de figuras

Lista de tabelas

Sumário

Introdução

O avanço do mundo digital trouxe consigo inúmeros impactos nos mais variados ramos da atuação humana, desde uma plantação a produção em larga escala, telecomunicações, meios de transporte, medicina, pesquisas científicas, relações sociais (globalização), segurança nacional, educação, etc.

Tal avanço está em correspondência biunívoca com o desenvolvimento social. As necessidades da sociedade implicam surgimentos de novas tecnologias que por sua vez implicam novas formas sociais de interagir, se organizar e estruturar, como exemplo, temos as práticas de ensino nas mais variadas escolas.

Esse desenvolvimento tecnológico goza de uma velocidade maior que as mudanças ocorridas em uma escola, principalmente durante e após a guerra fria, onde uma bipolarização mundial acelerou o surgimento e difusão de tecnologias (computadores, internet, etc). De uma maneira geral, a escola não acompanhou tal ritmo, pois a mesma necessita de um aperfeiçoamento tanto de pessoas como de infra estruturas, processos que podem ser demorados, principalmente na realidade das escolas públicas brasileiras, onde o financiamento de formação de pessoal e investimento em laboratórios ainda não é ideal.

Neste sentido uma questão que torna-se pertinente é : como mediar tal avanço na busca de uma nova educação, novas práticas, novas possibilidades?

Tal questionamento vai de encontro com um dos objetivos do Regimento 2017 do PROFMAT (Mestrado Profissional em Matemática em Rede Nacional) do capítulo I Art. 2º.

> Art. 2º O PROFMAT tem como objetivo proporcionar formação matemática aprofundada e relevante ao exercício da docência na Educação Básica, visando dar ao egresso a qualificação certificada para o exercício da profissão de professor de Matemática (SILVA, 2016, p.1).

As mudanças dentro do ambiente escolar são complexas pois demandam uma harmonia entre docentes, discentes, gestão e comunidade.

> As mudanças demorarão mais do que alguns pensam, porque os modelos tradicionais estão muito sedimentados, em parte, eles funcionam, e com isso torna-se complicado fazer mudanças profundas. Por outro lado encontramo-nos em processos desiguais de aprendizagem e evolução pessoal e social. Apesar de haver avanços, muitas instituições e muitos profissionais mantêm uma distância entre teoria e prática, entre suas ideias e ações. Se as pessoas têm dificuldades para evoluir, conviver e trabalhar em conjunto, isso se reflete na prática pedagógica (MORAN; MASETO; BEHRENS, 2013, pp.24-25).

As mudanças na educação passam em primeiro momento por educadores, não necessariamente professores, mas também os pais, diretores e os próprios alunos que estejam dispostos, maduros e que sejam motivadores, pois o gerenciamento das emoções afeta o rendimento de qualquer pessoa em sua aprendizagem.

A Educação focada na formação do ser humano, numa abordagem mais ampla possível, pode fazer uso de recursos tecnológicos que são um grande facilitador desse processo por diversos motivos:

1. Desafiam as instituições a sair do tradicional;
2. Facilitam a pesquisa, multiplicam os espaços de atuação ;
3. Possibilitam uma aprendizagem mais participativa e integrada;
4. Oferecem aplicativos, jogos, vídeos, dentre outros recursos, que proporcionam dinamismo nos processos educativos em geral.

É importante destacar que o fato de se ter um computador (ou qualquer outro recurso tecnológico), não acarreta que se tenha aprendizagem. O professor como mediador do conhecimento deve buscar e planejar atividades, com objetivos bem determinados que potencializem tais recursos.

Nesse contexto um fator marcante ao utilizarmos tecnologias como mediadores educacionais é o impacto explícito no espaço geográfico da sala de aula. Tal abordagem metodológica propicia uma maior integração entre ensino à distância e presencial, incentivando o nascimento de uma escola mais criativa e menos normativa e quem sabe mais libertária no sentido de que nem todos devam aprender as mesmas coisas, no mesmo tempo e submetidos às mesmas metodologias.

Toda sociedade será uma sociedade que aprende de inúmeras formas, em tempo real, com vastíssimo material audiovisual disponível.(MORAN; MASETO; BEHRENS, 2013, p.67)

O presente trabalho tem a proposta de relacionar o ensino de matemática (mais especificamente aritmética) com a utilização da linguagem de programação Python, pois é

cada vez mais recorrente o acesso a tecnologias e as transformações que estas possibilitam à sociedade. Nesse contexto as práticas de ensino de matemática podem tentar usufruir desses recursos tecnológicos para dar um maior dinamismo e tornar a aprendizagem mais prazerosa e significativa.

Por fim, este trabalho está estruturado conforme describo abaixo:

- Introdução : Tem por objetivo justificar a viabilidade e importânica da utilização de tecnologias na educação;
- Capítulo 1: apresenta de maneira breve como utilizar a linguagem Python e algumas de suas ferramentas;
- Capítulo 2: apresenta alguns programas em Python, elaborados a partir de ideias presentes em demonstrações clássicas de resultados de aritmética, exemplificando como a relação entre teoria e prática pode ser frutífera;
- Capítulo 3: traz as considerações finais, mostrando possibilidades de trabalhos futuros na busca de aprofundamento, tanto na linguagem Python como na sua utilização para o ensino da matemática.

Capítulo 1

Uma breve introdução ao Python

1.1 Porque Python

Dentre tantas opções em tecnologias, resolvemos trabalhar com a linguagem de programação Python por diversos motivos, os quais serão elencados abaixo:

"Python é, provavelmente, a linguagem de programação popular mais fácil e agradável de lidar. O código Python é simples para ler e escrever, e consegue ser conciso sem ser enigmático" (SUMMERFIELD, 2012, p.1).

> A linguagem de programação Phyton é muito interessante como primeira linguagem de programação devido a sua simplicidade e clareza. Embora simples, é também uma linguagem poderosa, podendo ser usada para administrar sistemas e desenvolver grandes projetos. É uma linguagem clara e objetiva, pois vai direto ao ponto, sem rodeios.(MENEZES, 2016, p.24)

Além do fato de Python ser um software livre (pode ser baixado gratuitamente) e multiplataforma (funciona nos sistema operacionais Windows, Linux e MacOS X, entre outros), é uma linguagem que vem crescendo nos últimos anos e é utilizadas por várias empresas. Além disso, é recomendada por diversos profissionais da área da computação e adodata por diversas empresas, como é ilustrado da Figura 1 a Figura 13.

Algumas empresas que usam o Python segundo (Python Brasil, 2018)

Figura 1 – Empresas que usam Python

Outras empresas que utilizam o Python de acordo com (Masanori, 2018)

Figura 2 – Outras empresas que usam Python

Figura 3 – Aplicativos que usam Python

Figura 4 – Instituições que usam Python

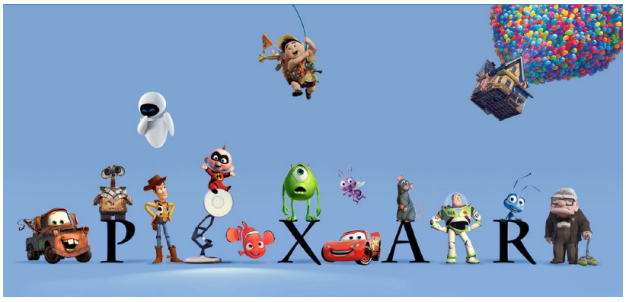


Figura 5 – Pixar usa Python

1.2 Instalação e ambientação

Primeiramente vamos mostrar como baixar e instalar o Python no sistema operacional Windowns 7, utilizando o navegador Google Chrome. Acessemos (Python ORG, 2018c)

Nessa tela da Figura 14, passemos o mouse no campo “Downloads”, depois cliquemos em “Python 3.6.4”, que é a versão disponível no dia 12/01/2018. Após o clique, automaticamente começará o download do Python como indicam as Figuras 15 e 16.

Após a conclusão do download, cliquemos no indicador de download da Figura 16. Então será aberta a janela de execução conforme Figura 17. Cliquemos em “executar”, após esse passo será aberta a janela do instalador, cliquemos em “Install Now” de acordo com a Figura 18, então o interpretador do Python começará a ser instalado (Figura 19). Terminada a instalação aparecerá o botão close, ao clicar em tal botão o interpretador da linguagem Python estará instalado no computador.

*"**Entre todas as linguagens que eu aprendi, Python é a que menos interfere entre mim e o problema. É a mais efetiva para traduzir pensamentos em ações**".*
***Eric Raymond**, Autor "A catedral e o Bazar".*
Contribuidor do GNU Emac, Linux, Fetchmail.
Mantém o Jargonfile, mais conhecido como "Dicionário dos Hackers"

Figura 6 – Comentário de Eric Raymond

*"**Python tem sido uma parte importante do Google desde o início**, e permanece assim conforme o sistema cresce e evolui... estamos procurando por mais pessoas com conhecimento nessa linguagem".*
***Peter Norvig**, diretor de qualidade de busca do Google Inc.*

Figura 7 – Comentário de Peter Norvig

Para iniciar o Python, no Windows 7, basta seguir a sequência de comandos: **Menu iniciar → Todos os programas → Python 3.6 → IDLE (Python 3.6 32-bit)**, conforme ilustra a Figura 20.

*"**Python é rápido o suficiente para o nosso site** e nos permite produzir características de fácil manutenção em tempos recordes, com um mínimo de desenvolvedores". **Cuong Do**, Software Architect, YouTube.com*

Figura 8 – Comentário de Cuon Dong

*"**Python é uma das cinco mais importantes linguagens que todo programador deve conhecer"** Bjarne Stroustrup, criador de C++*

Figura 9 – Comentário de Bjarne Stroutrup

Uma janela será aberta (Figura 21), descrita pelo interpretador como Python 3.6.4 Shell e nela é possível construir nossos programas em Python. Façamos nosso primeiro programa.

Digite: print("Bem vindo ao Python") e depois aperte enter.

"Only ugly languages become popular. **Python is the one exception**"
Don Knuth, walking to dinner after Alan Turing's Centenary Celebration
(from @ivanov on Twitter)

Figura 10 – Comentário de Donald Knuth

Figura 11 – Guido Van Rossum: Criador do Python

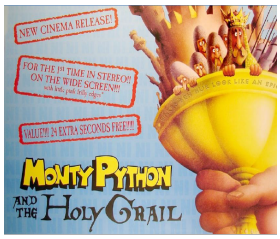


Figura 12 – Monty Python, grupo de humor que inspirou o nome linguagem

Figura 13 – Logomarca do Python

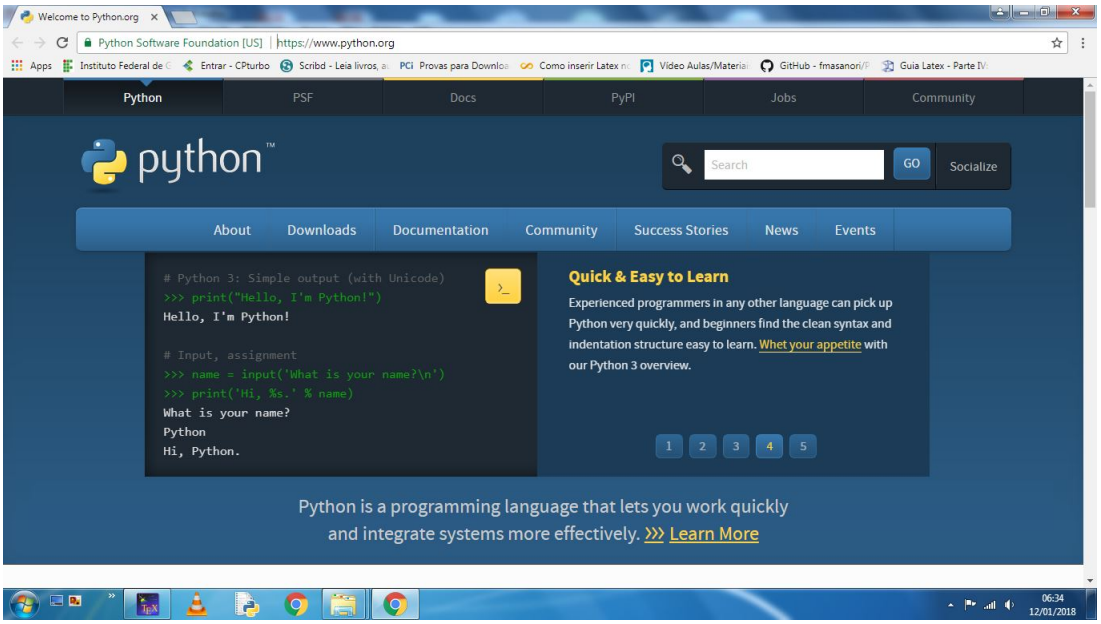


Figura 14 – Sítio para download do Python

O comando **print**(“Texto”) permite imprimir a palavra Texto na tela.

Código 1.1 – Primeiro programa

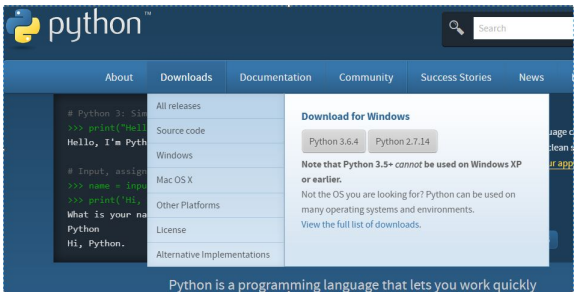


Figura 15 – Download do Python

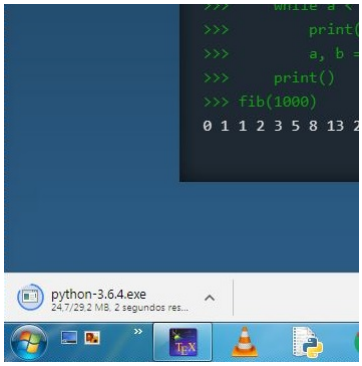


Figura 16 – Indicador do download no Google Chrome

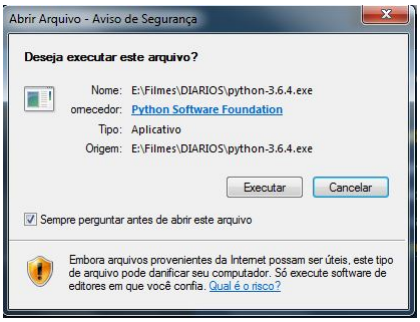


Figura 17 – Janela de confirmação da execução do instalador Python

Figura 18 – Janela de seleção das configurações

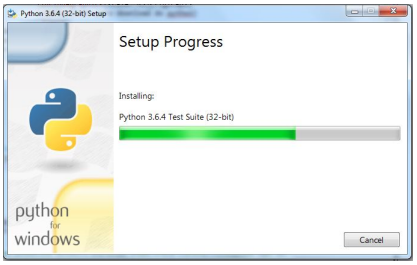


Figura 19 – Progresso da instalação

```
1 Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900
     32 bit (Intel)] on win32
2 Type "copyright", "credits" or "license()" for more information.
3 >>> print("Bem vindo ao Python")
4 Bem vindo ao Python
```

Observe que antes do **print** temos >>>, isto indica que estamos no modo interativo do Python (Python Shell), ou seja os comandos são executados logo após o enter. O modo interativo se apresenta por padrão ao abrirmos o interpretador. Mas nossos programas serão digitados no modo de edição. Para isso, no Python Shell, basta clicar em **File→New File** (Figura 22) dando acesso ao modo de edição, conforme Figura 23.

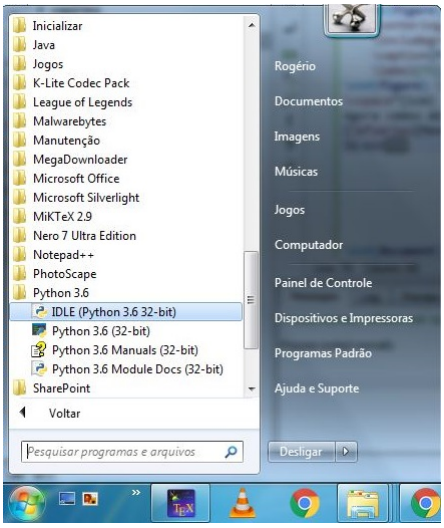


Figura 20 – Janela de confirmação da execução do instalador Python

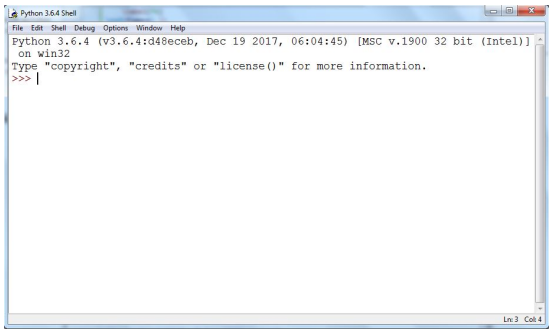


Figura 21 – Python Shell

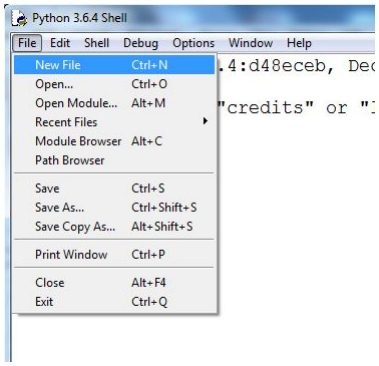


Figura 22 – Caminho para o modo de edição

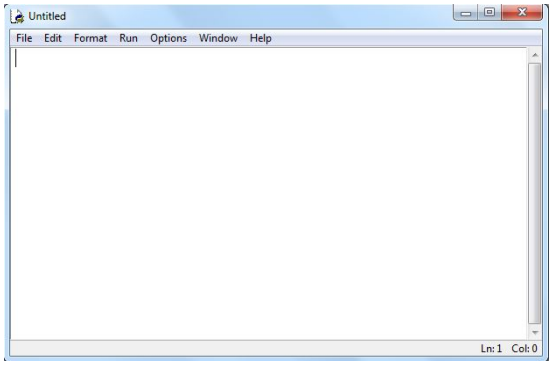


Figura 23 – Modo de edição do Python

Façamos o mesmo programa do Código 1.1 no modo de edição:

Código 1.2 – Primeiro Programa Modo de Edição

```
1 print("Bem vindo ao Python")
```

Depois de digitar **print**(“Bem vindo ao python”) e apertar enter não acontecerá nada. A fim de executarmos tal programa primeiro é necessário salvá-lo, seguindo os passos: **Run** → **Run module** → **Ok** → **nome do arquivo** → **Salvar** ; após isso, instantaneamente aparecerá uma janela (modo interativo) mostrando o resultado do seu programa. Tal procedimento é ilustrado conforme Figuras 24, 25, 26 e 27.

Por fim nossos programas terão o seguinte aspecto visual como o do Código 1.3 durante todo o trabalho. Tudo o que for digitado após # será ignorado pelo Python, sendo considerado um comentário.

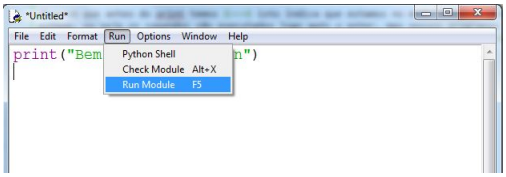


Figura 24 – Caminho para executar o programa no modo de edição

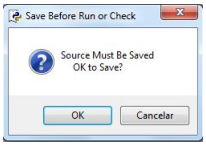


Figura 25 – "Ok" para salvar

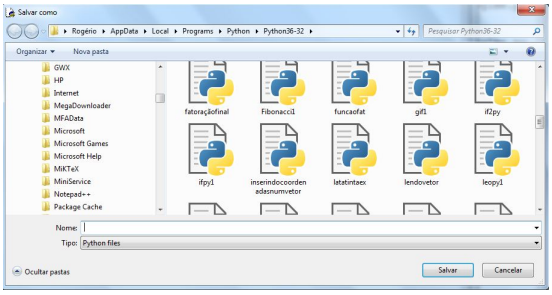


Figura 26 – Nomeando o arquivo

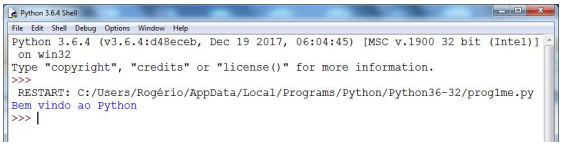


Figura 27 – Resultado do programa

Código 1.3 – Programa e resultado

```
1 print("Bem vindo ao Python")
2 RESTART: # Comentário: Abaixo segue o resultado do programa
3 Bem vindo ao Python
```

1.3 Python como calculadora

Podemos utilizar o Python como uma calculadora bem poderosa, pois suporta números tão grandes quanto a memória do computador pode aguentar. Os símbolos das operações básicas no Python são $+,-,*,/,**,//$ e $\%$. Dados a, $b \in \mathbb{R}$, vemos, na Tabela 1, o que cada um desses símbolos retorna quando acionado com os números a e b. Já no Código 1.4, vemos como usar o Python como calculadora.

Tabela 1 – Operações básicas

Símbolo	Comando	Retorna:
$+$	$a+b$	A soma
$-$	$a-b$	A subtração
$*$	$a*b$	A multiplicação
$/$	a/b	A divisão
$**$	$a**b$	A potência
$//$	$a//b$	O quociente em $\mathbb{Z}$
$\%$	$a\%b$	A resto da divisão em $\mathbb{Z}$

Código 1.4 – Operações Matemáticas

```
1 >>> 16+5 #  Após apertar Enter teremos exibido na tela a soma.
2 21
```

```
>>> 16 - 5 # A subtração.
11
>>> 16*5 # O asterisco *, retorna a multiplicação.
80
>>> 16**5 # Os ** representam a potência, aqui 16 está elevado a
    5.
1048576
>>> 16/5 # A barra (/) retorna a divisão.
3.2
>>> 16//5 # Duas (//) retornam o quociente da divisão inteira.
3
>>> 16%5 # O símbolo (%) (porcentagem) retorna o resto da divisão
    inteira.
1
```

É importante digitar os comandos apresentados no Código 1.4 com outros valores. Uma opção para efetuar os Códigos é acessando (Python ORG, 2018a), que nos encaminhará para uma versão do Python Shell online, caso se tenha conexão com a internet, proporcionando uma aprendizagem dinâmica em tempo real concomitante à leitura.

Caso se queira efetuar cálculos com valores, digamos mais específicos, como: $\sqrt{2}$, π, e, temos que importar, como no código abaixo, a biblioteca *math*, tal biblioteca nos dá acesso a uma boa gama de conteúdos matemáticos. Para maiores detalhes, consultar (MENEZES, 2016; SUMMERFIELD, 2012; Python ORG, 2018b).

Código 1.5 – $2^{\sqrt{2}}$, π^e e e^{π}

```
>>> import math # importando a biblioteca
>>> math.sqrt(2) # A raíz quadrada de 2.
1.4142135623730951
>>> math.pi # A constante pi
3.141592653589793
>>> math.e # A constante e
2.718281828459045
>>> 2**(math.sqrt(2)) # Potência com expoente irracional
2.665144142690225
>>> math.pi**(math.e) # Pi elevado a e
22.45915771836104
>>> math.e**(math.pi) # e elevado a pi
23.140692632779263
```

Ainda explorando os recursos do Python como calculadora, podemos resolver expressões numéricas e até calcular potências relativamente grandes, como no Código 1.6:

Código 1.6 – Expressões numéricas e 7^{1000}

```
>>> 2**3+12/4*2-10 # Uma expressão aritmética simples
4.0
>>> 2**3+12/(4*2-10) # Observe o uso do parênteses, alterando o
    resultado
2.0
>>> 2**3+12/4*(2-10)
-16.0
>>> 7**1000 # Cálculo de uma potência grande, cujo cálculo manual
    é inviável
1253256639965718318107554832382734206164985075080986171463495007
5209705963173811643244883905435152076319861591955159407668582898
9467263022761790838270854579830015111246661203984624358929832571
6157180147040963056680975076132736630232268952505413859271584260
8868449408241676861770818959228693603992231112568371921504668915
6738352590137241554510185855964549927575493247391132548534378497
9788060849510858742020118363623157274201095547829887915300882897
1184455050023048563841318994713214224394733419925930073562249293
7419453650061490302105127920314430401636855677549136337481321811
3496784270760914373450453993373486112611680559293554029928231924
9119036002703611228318093587277521451746401317827465710073632156
4606838252739601156414628445543663144696050650160812621814327062
6661951727017802002866450238230831859280613713103008292840711412
07731280600001
```

1.4 Variáveis

Até agora estávamos digitando explicitamente os números e realizando cálculos com os mesmo através do Python, no entanto em termos de Código é bem melhor utilizarmos uma variável (que pode ser uma letra ou um outro nome [1] qualquer).
O conteúdo de uma variável pode ser do tipo, inteiro (int), decimal (float), palavra (string), vetor (lista), etc. A vantagem de se utilizar variáveis é que podemos manipulá-las dentro de um código (programa) e deixar explícita a lógica de programação por trás do algoritmo.

Vejamos exemplos de variáveis numéricas no Python Shell no Código 1.7.

[1] o nome deve obrigatoriamente começar com uma letra, podem conter números e o símbolo _

Código 1.7 – Declarando e manipulando variáveis

```
>>> a = 3 # Declarando a variável a, e atribuindo-lhe o valor 3
>>> a
3
>>> 7+a # Operando com a
10
>>> 6*a # Outra operação com a
18
>>> b=10 # Declarando b
>>> b
10
>>> a+b # Operando com a e b
13
>>> a*b
30
>>> a/b
0.3
>>> c=2*a+3*b # Definindo c em função de a e b
>>> c
36
>>> a+b+c
49
```

Nos exemplos ilustrados no Código 1.7, atribuímos valores numéricos às variáveis. Contudo, veremos que também é possível atribuir outros "objetos" às variáveis. Por exemplo é possível atribuir uma palavra ou cadeia de caracteres (string) a determinada variável. É importante ressaltar que uma cadeia de caracteres no Python sempre deve estar especificada entre aspas duplas ou simples.

Vejamos no Código 1.8 uma variável do tipo cadeia de caractere e a função, **print** (já usada no Código 1.1). O Python reconhece que a função **print** deve ser executada, mostrando na tela o que estava em seu argumento (ou seja o que estiver dentro dos parênteses).

Código 1.8 – Variável do tipo string

```
>>> x="Olá Python" # Declarando x do tipo cadeia de caractere
>>> print(x) # A função print imprimirá o conteúdo da variável x
Olá Python
```

Além de enviarmos mensagens, podemos operar com palavras: acessar qualquer dos caracteres que a compõem, concatenar, fatiar e compor.

Supondo que temos uma variável x do tipo palavra, podemos saber o tamanho

(quantidade de caracteres) de x através da função **len**[2], logo **len**(x)="tamanho de x". É possível também acessar cada caractere da palavra x de acordo com a sua posição i por meio do comando $x[i]$. Ressaltamos que a primeira posição é interpretada como posição zero, caso a leitura da palavra seja da esquerda para a direita, com isso $0 \leq i \leq$**len**(x)-1. Caso leia-se x da direita para a esquerda os índices i serão negativos, com $-$**len**(x)$\leq i \leq -1$. Por exemplo, considere atribuída a palavra matemática à variável x, x='matemática'. Então, **len**(x)=10, $x[3]$ = e, $x[9]$ = a e $x[-2]$ = c. A figura abaixo apresenta a posição de todos os caracteres:

	$x[-10]$	$x[-9]$	$x[-8]$	$x[-7]$	$x[-6]$	$x[-5]$	$x[-4]$	$x[-3]$	$x[-2]$	$x[-1]$
x =	m	a	t	e	m	á	t	i	c	a
	$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$x[6]$	$x[7]$	$x[8]$	$x[9]$

Outra operação possível com uma variável x do tipo cadeia de caractere é o fatiamento. Por exemplo, o comando $x[i : j]$ fatia a palavra x, nos retornando seus caracteres entre as posições i e $j-1$, incluindo os caracteres nestas posições. Na tabela a seguir temos alguns dos comandos de fatiamento:

Tabela 2 – Fatiamento

Comando	Retorna a fatia
$x[i : j]$	de $x[i]$ até $x[j-1]$
$x[i :]$	de $x[i]$ até o último caractere
$x[: j]$	do primeiro até $x[j-1]$
$x[:]$	toda a variável x
$x[:: -1]$	x de trás pra frente
$x[:: 2]$	de $x[0]$ pra frente de 2 em 2
$x[:: -2]$	de $x[-1]$ pra trás de 2 em 2

Supondo agora que tenhamos duas variáveis x e y do tipo cadeia de caractere, a operação $x + y$ retorna a concatenação de x com y.

Abaixo um Código para melhor esclarecimento de tais operações.

Código 1.9 – Concatenação e Fatiamento

```
1 >>> x='matemática' # Declarando x variável tipo string
2 >>> y="computação" # Declarando y
3 >>> len(x) # A função len retorna o tamanho de x
4 10
5 >>> print(y[3]) # Acessando o caractere da terceira posição de y
6 p
```

[2] A função **len** só se aplica a variáveis tipo string ou lista

```
>>> print(x+y) # "Soma" de x com y, que é interpretado como a
    concatenação de x com y
matemáticacomputação
>>> print(y+x)
computaçãomatemática
>>> x[0:4] # Fatia de x que vai de x[0] até x[3]
'mate'
>>> y[6:] # Fatia de y que vai de y[6] até y[9] (último caractere)
'ação'
>>> x[::-1] # Esse comando retorna x de trás pra frente
'acitámetam'
>>> y[-10:-5] # Fatia que vai de y[-10] até y[-6]
'compu'
>>> x[::2] # Retorna de x[0] até o último caractere de 2 em 2.
'mtmtc'
>>> x[::-2] # Percorre de trás pra frente de 2 em 2, começando em
    x[-1]
'aiáea'
```

Finalmente, temos a composição que é uma operação que permite compor ou inserir numa mensagem variáveis designadas através de marcações. Por exemplo, na mensagem: **print**('Rogério tem %d anos.'%x), temos que o símbolo %d é marcador dentro da mensagem; tal marca será substituída pelo valor da variável x. No Python temos os marcadores: %d para inteiros, %f para decimais (float) e %s para palavras (strings).

No Código 1.10 apresentamos exemplos do uso de marcações dentro de um programa.

Código 1.10 – Marcadores

```
>>> idade=38 # declarando a variável idade do tipo inteiro
>>> print("Rogério tem %d anos"%idade)# O %d será trocado pelo
    valor de idade, essa é a composição.
Rogério tem 38 anos
>>> fruta='melancia' # declarando a variável fruta do tipo string
>>> print("Rogério tem %d anos e gosta de %s"%(idade,fruta))
Rogério tem 38 anos e gosta de melancia
>>> # no exemplo acima %s foi trocado pelo "valor" de fruta
>>> nome='Rogério'
>>> qtde=5
>>> vu=1.80
>>> print('%s comprou %d canetas por R$
    %4.2f'%(nome,qtde,qtde*vu))
Rogério comprou 5 canetas por R$ 9.00
>>> # O marcador %s foi trocado pelo valor da variável nome
```

```
>>> # O marcador %d pelo valor de qtde
>>> # O marcador %4.2f foi trocado pelo valor de 5*(1.80)
>>> # No %4.2f, o 4 significa 4 caracteres, 2 casa decimais e f
    float
```

Tratemos agora as variáveis lógicas ou booleanas que são True (verdadeiro) e False (falso). Podemos declarar uma variável como True ou False, mas geralmente estamos interessados em realizar comparações lógicas, com o uso de operadores relacionais. O resultado dessas comparações são sempre um valor lógico (True ou False) que depende da proposição especificada. Os operadores relacionais do Python estão elencados na Tabela 3, ilustrados no Código 1.11.

Tabela 3 – Operadores relacionais

Símbolo	Comparação (Pergunta)
==	igualdade
!=	diferente
>	maior do que
<	menor do que
>=	maior ou igual
<=	menor ou igual

Código 1.11 – Operadores relacionais

```
>>> a=True # Declarando a do tipo lógico (booleana)
>>> b=False
>>> c=5
>>> d=10
>>> a==b # Perguntando se a é igual a b.
False
>>> a!=b # Perguntando se a é diferente de b.
True
>>> c==7
False
>>> c<d # Comparando c com d, neste caso, perguntando se c é
    menor que b.
True
>>> d >= 10 # Comparando b com 10; neste caso, perguntando se é
    maior ou igual a 10.
True
>>> d > 10 # Perguntando se d é maior que 10.
False
>>> # Pode-se escrever a=2 ou a = 2 o Python entende as duas
    formas
```

No Python, também é possível especificar operadores lógicos, que são: **not** (não), **and** (e) e **or** (ou), conhecidos também como negação ($\sim$ ou $\neg$), conjunção ($\wedge$) e disjunção ($\vee$), respectivamente.

Sejam p e q proposições quaisquer. Pode-se formar outras proposições envolvendo p, q e os operadores lógicos. Por exemplo: $\neg p$ (**not** p), $p \wedge q$ (p **and** q) e $p \vee q$ (p **or** q).

As Tabelas 4, 5 e 6 apresentam os valores de verdade das proposições compostas envolvendo proposições p e q a partir dos valores de verdade de p e de q e do operador utilizado na construção da proposição. O Código 1.12 exemplifica o uso de tais operadores.

Tabela 4 – **not**

p	**not** p
V	F
F	V

Tabela 5 – **and**

p	q	p **and** q
V	V	V
V	F	F
F	V	F
F	F	F

Tabela 6 – **or**

p	q	p **or** q
V	V	V
V	F	V
F	V	V
F	F	F

Código 1.12 – Operadores lógicos

```
>>> p=True
>>> q=False
>>> c=5
>>> d=10
>>> not p # A negação de p
False
>>> p or q # A proposição p ou q
True
>>> c!=5 and d==10 # Proposição composta usando o and, será falsa
    de acordo com a Tabela 5.
False
>>> c<5 or 2*d > 18 # Proposição composta usando o or, será
    verdadeira de acordo com a Tabela 6.
True
>>> (d%c==0 and d>=10) or (not (p and q)) # Uma proposição mais
    complexa (expressão lógica)
True
```

Observe que a última expressão lógica do Código 1.12 é um pouco mais complexa; e existe uma ordem de prioridade a ser obedecida para determinar seu valor lógico: primeiro os (), segundo o **not**, terceiro o **and** e por último o **or**.

Diante de tal hierarquia resolvamos a expressão lógica:

(d%c==0 **and** d>=10) **or** (**not** (p **and** q)) com p=True, q=False, c=5 e d=10

Como d%c==0 é Verdadeira (V) pois 10 é divisível por 5, d>=10 é V, pois d=10, logo (d%c==0 **and** d>=10) é V.

Como p **and** q é F, logo (**not** (p **and** q)) é V

Finalmente nossa expressão se resume em V **or** V que é V.

De modo simplificado:

(d%c==0	**and**	d>=10)	**or**	(**not**	(p	**and**	q))
(V	**and**	V)	**or**	(**not**	(V	**and**	F))
		V	**or**	(**not**	F)		
		V	**or**	V			
			V				

Um outro tipo de variável que usaremos serão as listas ou vetores. Podemos entender uma lista como os vagões de um trem, sendo que uma lista poder ter zero ou mais vagões e dentro dos mesmos seus elementos.

Uma lista L de n elementos é representada por L=[x_0,x_1,...,x_{n-1}], em que os colchetes servem para indicar que L é uma lista. Uma lista vazia em Python é representada por L=[ ]. Os elementos x_i de L podem ser palavras (strings), números ou até mesmo outra lista. Por exemplo, seja L a lista com as vogais, ou seja, L=['a','e','i','o','u']. Os elementos de L são letras, mas como dito anteriormente, podemos ter elementos de outros tipos, como, por exemplo, na lista W=[1,2,'x','y',['a','b']] os dois primeiros elementos são números inteiros, o terceiro e quarto são letras e o último é outra lista.

Seja L uma lista. Assim como nas palavras (strings), podemos usar a função **len** para descobrirmos o tamanho de L, podemos acessar qualquer elemento de L com o comando L[i] , $0 \leq i \leq$ **len**(L) -1 , e ainda, podemos fatiar L exatamente como na Tabela 1, ou seja, L[i:j] é uma fatia de L que vai de L[i] até L[$j-1$].

Agora, especificamente para listas, podemos modificar qualquer um de seus elementos, atribuindo novo valor ao elemento i, por meio do comando L[i]= novo valor. Além disso, podemos acrescentar um vagão ao final de uma lista, através do método append. De um modo geral o método append funciona da seguinte maneira: se L=[x_0,x_1,...,x_{n-1}] uma lista com n elementos, ao digitarmos L.append(x_n), L se modificará para L=[x_0,x_1,...,x_{n-1}, x_n] passando a ter $n+1$ elementos.

Dadas listas L e W, L+W será uma nova lista começando no primeiro elemento de L e terminando no último de W ordenadamente. Ressaltamos ainda que n*L designa a lista $\underbrace{\text{L}+\text{L}+\ldots+\text{L}}_{n \text{ vezes}}$. Abaixo segue um Código utilizando todos os recursos descritos acima.

Código 1.13 – Listas e algumas operações

```
>>> L=['a','b','c','d','e'] # declarando L, variável tipo lista (vetor)
>>> L
['a', 'b', 'c', 'd', 'e']
>>> len(L) # A função len retorna o tamanho da lista
5
>>> L[2] # Aqui estamos acessando um elemento da lista (lembre-se que a contagem das posiçoes inicia-se em zero)
'c'
>>> L[2]=15 # Atribuindo um novo valor a posição 2 de L
>>> L
['a', 'b', 15, 'd', 'e']
>>> L.append('f') # O append insere uma nova entrada ao final de L, cujo conteúdo é f.
>>> L # Agora L terá tamanho 6 e L[5]='f'
['a', 'b', 15, 'd', 'e', 'f']
>>> print(L+['g','h']) # Somando listas
['a', 'b', 15, 'd', 'e', 'f', 'g', 'h']
>>> print(2*[3,4]) # Multiplicando uma lista por 2
[3, 4, 3, 4]
>>> L[2:5] # Fatia de L da segunda até a quarta posição
[15, 'd', 'e']
>>> L[::-1] # Invertendo a lista
['h', 'g', 'f', 'e', 'd', 15, 'b', 'a']
```

1.5 Entrada de Dados

Até aqui nossos pequenos programas já vinham com valores fornecidos previamente para as variáveis. No entanto podemos solicitar que um usuário forneça dados (através do teclado por exemplo), a fim de executar um programa diversas vezes, sem alterar sua estrutura. A função que permite a entrada de dados é a **input**. Sua sintaxe é **input**('mensagem solicitando ao usuário um dado')

Vejamos um exemplo acerca de entrada de dados no Código 1.14 .

Código 1.14 – Entrada de Dados

```
x = input("Digite um número: ") # x terá o valor que for digitado
print(x)
RESTART: # Após o RESTART temos a execução do programa
Digite um número:56
56
```

No programa do Código 1.14 tem-se que digitar um número e depois apertar a tecla enter (caso contrário o programa não é executado); no exemplo, foi digitado o 56.

Prosseguindo, vejamos outro programa com entrada de dados, no Código 1.15.

Código 1.15 – Input e marcadores

```
1 nome = input("Digite seu nome: ") # Aqui a variável nome que
     depende da entrada
2 print("Muito prazer %s, bons estudos."%nome)
3 RESTART:
4 Digite seu nome: Edilene
5 Muito prazer Edilene, bons estudos.
```

É importante destacar o seguinte detalhe, se digitássemos no Programa 1.14 acima, no lugar de número a palavra olá e no Programa 1.15 no lugar de nome o número 123, por exemplo, os programas rodariam perfeitamente (se possível, faça o teste). Isto ocorre porque a função **input** sempre retorna um valor do tipo cadeia de caractere (string) e para contornar tal situação devemos converter a entrada de dados de acordo com o tipo que queremos para a variável. O Programa 1.16 exemplifica a situação descrita acima.

Código 1.16 – Entrada sem conversão

```
1 a=input("Digite o primeiro número: ")
2 b=input("Digite o segundo número: ")
3 print(a+b)
4 RESTART:  # O print retornou a concatenação de 40 com 12
5 Digite o primeiro número: 40
6 Digite o segundo número: 12
7 4012
```

A conversão da entrada de dados é algo bem interessante e bastante funcional.Vejamos algumas conversões na tabela abaixo:

Tabela 7 – Conversão de dados

Comando	Entrada	Retorna
x=**int**(**input**('nº: '))	23	23 tipo inteiro
x=**float**(**input**('nº: '))	3.14	3.14 tipo float(decimal)
x=**str**(123)	123	'123' tipo string(caractere)

Nosso próximo Código é bem parecido com o Código 1.16, no entanto, com os dados convertidos

Código 1.17 – Entrada de dados com conversão

```
1 a = int(input("Digite o primeiro número: "))
2 b =int(input("Digite o segundo número: "))
3 print(a+b)
4 RESTART:  # O print nessa caso retorna a soma de 40 com 12
5 Digite o primeiro número: 40
6 Digite o segundo número: 12
7 52
```

Observe que a conversão dos dados de entrada no Código 1.17 permite que o Python entenda 40 e 12 como números, o que não ocorre no Código 1.16.

Podemos também converter números em strings, caso o interesse seja aplicar um método específico para strings, para uma lista dos métodos aplicáveis a string consultar (SUMMERFIELD, 2012, p. 68-70).

1.6 Estruturas básicas

Todos os programas até aqui foram sequenciais, isto é, executaram todas as linhas uma após a outra. No entanto isso nem sempre será necessário. Por vezes pode-se estar interessado que uma parte do programa seja executada apenas se uma dada condição for verdadeira. Em Python a estrutura de decisão é o **if**.

O **if** é o nosso "se", que pode ser entendido da seguinte forma: Se a condição for verdadeira faça algo. Essa condição será uma expressão lógica e a estrutura do **if** será:

if<condição>**:**
comando 1
comando 2
⋮
comando n

Observe que após a condição do **if** temos o símbolo :(dois pontos) e que os comandos de 1 até n estão todos alinhados e um pouco à direita e abaixo do **if**; este recuo à direita é chamado de **identação** e caracteriza o bloco de comando(s) do Python, nesse caso da estrutura **if**. Como dito anteriormente, tal bloco só será executado caso a condição do **if** seja verdadeira, caso contrário será ignorado na execução do programa.

O próximo Programa pedirá que o usuário forneça dois números e retornará uma mensagem dizendo qual foi o maior número.

Código 1.18 – Estrutura condicional **if**

```
1 a = int(input("Digite o primeiro número: "))
2 b = int(input("Digite o segundo número: "))
3 if a > b: # A linha 4 só será executada caso a>b seja verdadeiro
4     print('O primeiro número é o maior')
5 if b > a:
6     print('O segundo número é o maior')
7 RESTART: # Aqui é o resultado do programa já na janela interativa
8 Digite o primeiro número: 35
9 Digite o segundo número: 98
10 O segundo número é o maior
```

Vejamos outro Programa, usando **if**, que leia a velocidade de um carro, caso esteja acima de 80 km/h, calcule uma multa de R$ 5,00 por km/h quando o veículo está nestas condições e emita uma mensagem dizendo que o carro foi multado e o respectivo valor de multa.

Código 1.19 – Estrutura condicional **if**

```
1 v = int(input('Qual a velocidade do carro? '))
2 if v > 80:
3     m = 5*(v-80)
4     print("Você foi multado, sua multa foi de %5.2f" %m)
5 RESTART:
6 Qual a velocidade do carro? 85
7 Você foi multado, sua multa foi de R$ 25.00
```

Pode-se usar vários **if**'s no mesmo Programa, no entanto quando temos uma condição complementar ao **if** podemos fazer uso de outra palavra reservada do Python que é o **else**. Por exemplo, no Programa 1.19 caso a velocidade seja menor ou igual a 80, podemos usar o **else** para emitir uma mensagem ao usuário. Tal situação é exemplificada no Programa 1.20.

Código 1.20 – Uso do **else**

```
1 v = int(input('Qual a velocidade do carro? '))
2 if v > 80:
3     m = 5*(v-80)
4     print("Você foi multado, sua multa foi de %5.2f" %m)
5 else:
6     print("Parabéns! Você está no limite permitido")
7 RESTART:
8 Qual a velocidade do carro? 80
9 Parabéns! Você está no limite permitido
```

Outra estrutura muito importante é a de repetição (laço), que é usada para executar a mesma parte de um programa várias vezes enquanto uma condição for verdadeira. Uma das estruturas de repetição do Python é o **while**, cujo formato é:

while<condição>**:**
comando 1
comando 2
⋮
comando n

Por exemplo, se quisermos um Programa que imprima todos os números naturais de 1 até 10, podemos utilizar um **while**, como no Programa 1.21:

Código 1.21 – Estrutura **while**

```
1  x=1 # declarando a variável x e atribuindo-lhe valor 1
2  while x<=10: # As linhas 3 e 4 serão executadas enquanto x<=10
3      print(x)
4      x=x+1 # aqui um novo conceito, o contador, incrementa-se a x
       uma unidade a cada execução
5  RESTART:
6  1
7  2
8  3
9  4
10 5
11 6
12 7
13 8
14 9
15 10
```

Observe que x, a cada vez que o laço se repetia, era acrescido do valor constante 1; quando isso ocorre com uma variável em um programa, tal variável recebe o nome de contador.

Quando o valor que uma variável vai recebendo durante a execução de um programa varia, estamos diante de um acumulador.

Observe também que os valores impressos na saída estão na vertical, isto é, o **print** imprime um número e salta uma linha, até o laço findar. Caso queiramos apresentar a saída dos números na horizontal, podemos criar um vetor para receber cada valor de x e depois imprimir tal vetor conforme Código 1.22.

Código 1.22 – Programa usando **while** com vetor de saída

```
x=1
números=[]
while x<=10:
    números.append(x)
    x=x+1
print(números)
RESTART:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Uma outra alternativa seria usar o comando **print**(x, end=“ ”). Tal comando imprime todos os valores de x gerados no laço da estrutura de repetição em linha horizontal, como ilustra o código abaixo:

Código 1.23 – Imprimindo em linha

```
x=1
while x<=10:
    print(x, end=" ")
    x=x+1
RESTART:
1 2 3 4 5 6 7 8 9 10
```

Podemos fazer variações do Programa 1.22 como: listar todos os números pares (ou ímpares) até 30, ou até um dado $n \in \mathbb{N}$ fornecido pelo usuário; exibir apenas os múltiplos de 3, dentre outras. Enfim, são várias as possibilidades tanto usando o **if** quanto o **while** que propiciam atacar vários problemas, buscando um programa que os solucione.

O Programa 1.24 a seguir, pedirá ao usuário que forneça um número (natural) e exibirá todos os números pares até o número fornecido.

Código 1.24 – Pares até um certo n

```
n=int(input('Digite um número: '))
x=0
pares = []
while x<=n:
    pares.append(x)
    x=x+2
print('Todos os pares até %d são'%n, pares)
RESTART: Abaixo uma execução quando o n = 10
Digite um número: 10
Todos os pares até 10 são [0, 2, 4, 6, 8, 10]
```

Para efeito de treinamento, façamos um programa que leia 10 números fornecidos pelo usuário e retorne a média dos 10 números. Nesse programa serão usados tanto contador quanto acumulador.

Código 1.25 – Média de 10 números fornecidos pelo usuário

```
n = 1 # Aqui nosso contador
s = 0 # Aqui nosso acumulador
while n<=10:
    x=int(input("Digite o %d° número: "%n))
    s=s+x
    n=n+1
print("A média dos 10 números fornecidos é %5.2f"%(s/10))
RESTART:
Digite o 1° número: 1
Digite o 2° número: 2
Digite o 3° número: 3
Digite o 4° número: 4
Digite o 5° número: 5
Digite o 6° número: 6
Digite o 7° número: 7
Digite o 8° número: 8
Digite o 9° número: 9
Digite o 10° número: 10
A média dos 10 números fornecidos é  5.50
```

Observe que a variável s acumula (somando) todos os 10 valores inseridos pelo usuário, enquanto que a variável n vai incrementando seu valor de 1 em 1, ou seja n é um contador.

Podemos entender que um contador é um caso particular de acumulador no qual acumula sempre um mesmo valor constante.

Às vezes é necessário colocar uma estrutura dentro de outra em um programa. Tal procedimento chama-se ”aninhar”; então em estruturas aninhadas temos um **if** dentro de outro **if**, ou um **while** dentro de outro **while**, ou um **if** dentro de um **while** e vice-versa.

Façamos um programa que calcule o valor da conta telefônica de acordo com os minutos gastos se a empresa oferece os seguintes planos: abaixo de 100 minutos cobra R$ 0,40 o minuto; entre 100 e 200 minutos cobra R$ 0,35 por minuto; acima de 200 minutos cobra R$ 0,20 por minuto.

Código 1.26 – Conta telefônica

```
min = int(input("Quantos minutos você utilizou esse mês?  "))
if min < 100:
```

```
3     p = 0.40
4 else:
5     if min < 200: # Esse if está aninhado (dentro) do else
6         p = 0.35
7     else: # esse else está aninhado no else da linha 4
8         p = 0.20
9 print("Sua conta esse mês será de R$ %6.2f"%(min*p))
10 RESTART:
11 Quantos minutos você utilizou esse mês?  150
12 Sua conta esse mês será de R$  52.50
```

Podemos também substituilr um par **else** e **if** pela palavra reservada **elif**. Esteticamente utilizar o **elif** é interessante quando o programa apresenta muitos pares **else** e **if**. O código a seguir é praticamente idêntico ao Código 1.26, exceto pelo uso do **elif**

Código 1.27 – Conta telefônica, uso do **elif**

```
1 min = int(input("Quantos minutos você utilizou esse mês?  "))
2 if min < 100:
3     p = 0.40
4 elif min < 200: # elif substituindo um else e if
5     p = 0.35
6 else:
7     p = 0.20
8 print("Sua conta esse mês será de R$ %6.2f"%(min*p))
9 RESTART:
10 Quantos minutos você utilizou esse mês?  150
11 Sua conta esse mês será de R$  52.50
```

No Código 1.28, temos mais um exemplo de estruturas aninhadas. Tal programa exibe a tabuada de multiplicação do 1 ao 10, para isso vamos usar um **while** dentro de um **while**.

Código 1.28 – Tabuada de Multiplicação

```
1 tabuada=1
2 while tabuada<=10:
3     número = 1
4     while número <=10: # while aninhado com while
5         print("%d x %d = %d"%(tabuada,número,tabuada*número))
6         número+=1 # abreviação de número=número+1
7     tabuada+=1
8 # A saída desse programa não será apresentada, rode-o e entenda o
    porquê.
```

Outra palavra reservada bem útil é o **break**. O **break** é uma instrução que força a parada do laço de um **while**. Por exemplo, se quisermos escrever um programa que imprima a soma dos números digitados por um usuário, onde tal usuário digita tantos números quanto queira, parando a soma apenas quando o 0 (zero) for digitado, daí temos o número 0 como condição de parada (break).

Código 1.29 – Soma dos n valores fornecidos, uso do **break**

```
s=0
while True: # esse while roda infinitamente
    n = int(input("Digite um número para somar ou 0 para sair: "))
    if n == 0: # um if aninhado dentro do while
        break
    s = s + n # s acumulando todos os valores digitados
print("A soma de todos os números digitados é %d"%s)
RESTART:
Digite um número para somar ou 0 para sair: 3
Digite um número para somar ou 0 para sair: 5
Digite um número para somar ou 0 para sair: 7
Digite um número para somar ou 0 para sair: 6
Digite um número para somar ou 0 para sair: 0
A soma de todos os números digitados é 21
```

Dando continuidade às estruturas básicas, temos no Python uma estrutura muito interessante quando queremos percorrer os elementos de uma lista (vetor), tal estrutura é o **for**.

Sua sintaxe é a seguinte:

for *nome da variável* **in** lista**:**
comando 1
comando 2
⋮
comando n

Ele funciona como se fosse um **while**; enquanto a variável está percorrendo os elementos da lista ele executa o bloco de comandos.

Vejamos um programa usando o **for** que imprima os cinco primeiros quadrados perfeitos :

Código 1.30 – Quadrados usando o **for**

```
for i in [1,2,3,4,5]: # a variável i assumirá os valores de 1 a 5
    print(i**2)
```

```
RESTART:
1
4
9
16
25
```

O **for** é muito interessante para lidarmos com listas númericas. No entanto, não é muito prático digitar explícitamente uma lista como no Programa 1.30.

Para gerar uma lista numérica vamos usar o comando **list**(**range**(n))=[0,1,2,...,n-1]. A Tabela 8 ilustra os comandos básicos das listas com range e o Código 1.31 exemplifica o uso de cada um destes comandos.

Tabela 8 – Listas numéricas com **list** e **range**

Comando	Retorna
list(**range**(n))	[0,1,2,...,n-1]
list(**range**(p,n))	[p,p+1,p+2,...,n-1]
list(**range**(p,n,k))	[p,p+k,p+2k,...,p+lk] com p+lk < n

Código 1.31 – Listas com **list** e **range**

```
v_1=list(range(10))
v_2=list(range(5,10))
v_3=list(range(1,10,2))
print(v_1,'\n',v_2,'\n',v_3) # O \n serve para pular linha dentro
    do print
RESTART:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9]
[1, 3, 5, 7, 9]
```

Utilizando o **for** podemos criar listas mais específicas, como os primeiros 10 cubos positivos, como no Código 1.32.

Código 1.32 – Dez primeiros cubos

```
cubos=[]
for i in range(1,11): # intervalo de 1 até 10
    cubos.append(i**3)
print('Os dez primeiros cubos positivos são: \n',cubos)
RESTART:
Os dez primeiros cubos positivos são:
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Encerraremos este capítulo com o conceito de funções. O Python já possui algumas funções pré-definidas, por exemplo **len**, **input**, **print**, dentre outras outras.

A ideia é criar outras funções, cujo processo funcionará basicamente da seguinte forma: definiremos uma função de uma ou mais variáveis e, dentro da definição da mesma, tem-se códigos que farão com que a função retorne algum valor (podendo ocorrer de criar-se funções que não precisem de parâmetros e/ou não retornem nada). Sua sintaxe é:

def *nome da função*(parametros):
comando 1
comando 2
⋮
comando n
return *expressão*

Vamos criar uma função que retorne a soma de dois números, conforme Código 1.33.

Código 1.33 – Função soma

```
1  def soma(x,y):
2      return x+y
3  print(soma(10,90))
4  RESTART:
5  100
6  >>> soma(5,11) # chamando a função interativamente
7  16
8  >>> soma(1.3,2.7)
9  4.0
10 >>> soma(-1,-8)
11 -9
```

Vejamos outra função que diz se um dado natural é par.

Código 1.34 – Função par

```
1  def epar(n):
2      if n%2==0:
3          return True
4      else:
5          return False
6  RESTART:
7  >>> epar(1)
8  False
9  >>> epar(4)
10 True
```

No próximo capítulo utilizaremos, conforme a necessidade, os comandos aqui apresentados com o intuito de resolver alguns problemas de matemática através de programas na linguagem Python.

Capítulo 2

Aplicações da Aritmética básica em Python

Vamos apresentar neste capítulo alguns resultados que envolvem números naturais (denotado por $\mathbb{N}$), no intuito de implementá-los com a linguagem Python. Apresentaremos por exemplo: a divisão euclidiana, o algoritmo de Euclides[1], o Crivo de Eratóstenes, dentre outros. Tais resultados, apesar de milenares, se mostram como algoritmos bastante eficazes quando implementados.

2.1 Divisão Euclidiana

Comecemos com a definição de divisibilidade entre dois números naturais.

Definição 2.1 (Divisibilidade)**.** *Dados a,b,$\in \mathbb{N}$, com $a \neq 0$, dizemos que a divide b se existir $c \in \mathbb{N}$ tal que $b = a \cdot c$. Denotamos a divide b por $a|b$.*

Por exemplo: $5|10$ pois existe $2 \in \mathbb{N}$ tal que $10 = 5 \cdot 2$. Quando $a|b$, diz-se que b é um múltiplo de a ou b é divisível por a.

Ocorre que nem sempre a dividirá b, mas pode-se sempre fazer a divisão de b por a deixando algum resto, este é o conteúdo do teorema a seguir, a Divisão Euclidiana, o qual Euclides já utilizava nos *Elementos*[2].

Antes de enunciar o Teorema da Divisão Euclidiana, vamos assumir como axioma o Princípio da Boa Ordem:

Axioma 2.1 (Boa Ordem)**.** *Todo subconjunto não vazio de $\mathbb{N}$ possui um menor elemento.*

[1] Matemático grego que viveu por volta de 300 a.C. e escritor da inestimável obra *Os Elementos*

[2] Uma das obras mais importantes da matemática, composta por 13 livros onde se reunia praticamente todo conhecimento matemático da época

Teorema 2.1 (Divisão Euclidiana)**.** *Dados $a, b \in \mathbb{N}$ com $0 < a < b$. Existem dois únicos naturais q e r tais que:*

$$b = aq + r \quad com \quad r < a$$

Demonstração. **Existência:** Considere a sequência S de números naturais

$$S = (b,\, b - a,\, \ldots,\, b - n \cdot a,\, \ldots)$$

Pelo Axioma 2.1 tal sequência não pode decrescer infinitamente, uma vez que caracteriza um subconjunto não vazio de $\mathbb{N}$; logo S possuirá um menor elemento r, digamos $r = b - q \cdot a$. Note que se $r < a$ teremos mostrado a existência de q e r como está no enunciado do Teorema.

De fato, $r < a$, pois caso $r > a$ teríamos $r - a > 0 \in S$; isto contraria o fato de r ser o mínimo de S. Logo $b = aq + r$ com $r < a$.

Unicidade: Suponha que existam r' e q' tais que $b = aq' + r'$ com $r' < a$. Segue que $r \leq r' < a$ e $0 \leq r' - r < a$. Da igualdade $0 = b - b = aq' + r' - (aq + r)$ temos que $(q - q')a = r' - r \Longrightarrow a | r' - r \Longrightarrow r' - r = 0$ já que $a > r' - r$. Portanto $r' = r$ e consequentemente $q' = q$. □

Por exemplo:

Caso $b = 37$ e $a = 7$ temos $q = 5$ e $r = 2$, pois $37 = 7 \cdot 5 + 2$;

Caso $b = 89$ e $a = 13$ temos $q = 6$ e $r = 11$, pois $89 = 13 \cdot 6 + 11$.

Observemos que a demonstração acima nos fornece um algoritmo para dividir dois números naturais, vejamos a implementação de um programa em Python que nos retorne o quociente e o resto da divisão de dois naturais fornecidos pelo usuário.

Código 2.1 – Divisão Euclidiana

```
1  b=int(input("digite o valor de b: "))
2  a=int(input("digite o valor de a: "))
3  q,r=0,0 # q e r são quociente e resto respectivamente
4  if b>=a:
5      while b-q*a>=a:
6          q=q+1
7          r=b-q*a
8    print("Na divisão de %d por %d, temos quociente %d e resto
     %d"%(b,a,q,r))
9  else:
10     print("Na divisão de %d por %d, temos quociente %d e resto
     %d"%(b,a,q,b))
11 RESTART:
```

```
digite o valor de b: 37
digite o valor de a: 7
Na divisão de 37 por 7, temos quociente 5 e resto 2
```

Alternativamente poderíamos calcular o quociente e o resto através das operações // e %, como no programa abaixo:

Código 2.2 – Divisão Euclidiana com operações

```
b=int(input('Digite o valor de b: '))
a=int(input('Digite o valor de a: '))
q=b//a
r=b%a
print('Na divisão de %d por %d, temos quociente %d e resto
   %d'%(b,a,q,r))
RESTART:
Digite o valor de b: 46
Digite o valor de a: 7
Na divisão de 46 por 7, temos quociente 6 e resto 4
```

Contudo, como pretende-se fazer uma conexão entre as demonstrações dos resultados e sua implementação, acreditamos que, didaticamente, o Código 2.1 é mais interessante.

2.2 Bases numéricas

Várias civilizações criaram maneiras de organizar a forma como representariam suas quantidades para atender diversas demandas. Tais maneiras são conhecidas como sistemas de numeração, dentre os quais podemos citar: Romano, Maia, Egípcio, dentre outros. Atualmente o sistema universalmente utilizado para representar números é o decimal posicional.

> Esse sistema de numeração que é uma variante do sistema sexagesimal utilizado pelos babilônios 1700 anos antes de Cristo, foi desenvolvido na China e na Índia. Existem documentos do século VI comprovando a utilização desse sistema. Posteriormente, foi se espalhando pelo Oriente Médio, por meio das caravanas, tendo encontrado grande aceitação entre os povos árabes. A introdução do sistema decimal na Europa foi tardia por causa dos preconceitos da Idade Média. Por exemplo, num documento de 1299, os banqueiros de Florença condenavam seu uso. (SIDKI, 1975, p.7)

No sistema decimal todo número é formado por uma sequência dos algarismos:

$$0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9$$

Como são dez algarismos daí o nome decimal (base 10), posicional pois a posição do algarismo na sequência que forma o número lhe atribui um peso (potência de 10) diferente.

Por exemplo, $23 = 2 \cdot 10^1 + 3 \cdot 10^0$ e $32 = 3 \cdot 10^1 + 2 \cdot 10^0$; portanto, a posição em que os números 2 e 3 se apresentam é importante.

De uma maneira geral, o número $b_n\, b_{n-1} \ldots b_2\, b_1\, b_0$ no sistema decimal é o mesmo que $b_n \cdot 10^n + b_{n-1} \cdot 10^{n-1} + \ldots + b_2 \cdot 10^2 + b_1 \cdot 10^1 + b_0 \cdot 10^0$ com $n \in \mathbb{N}$ e $0 \leq b_i \leq 9$.

Outro sistema que convém destacar é o binário (base 2) usado na computação. Futuramente, exemplificaremos tal sistema.

Tanto o sistema decimal quanto o binário são posicionais e de base constante. Na verdade, dados dois números a e $b > 1$ arbitrários podemos sempre escrever a na base b. O modo de fazê-lo se baseia no Teorema da Mudança de Base que é uma aplicação da divisão euclidiana. Contudo, para prová-lo, iremos fazer uso do Segundo Princípio de Indução que será demonstrado abaixo.

Teorema 2.2 (Segundo Princípio de Indução)**.** *Sejam $P(n)$ uma propriedade referente a números naturais, $n \in \mathbb{N}$, e a um número natural. Se $P(a)$ é verdadeira e, se para $a \leq k < n$, a validade de $P(k)$ implicar a veracidade de $P(n)$, então $P(n)$ será verdadeira para todo natural $n \geq a$.*

Demonstração. Considere o conjunto $V = \{n \in \mathbb{N}; P(n) \text{ verdadeira}\}$ queremos provar que $V = \mathbb{N}$. Suponha por absurdo que $V \neq \mathbb{N}$, daí $\mathbb{N} \backslash V \neq \emptyset$. Logo, pelo Axioma 2.1 existe o menor elemento $k \in \mathbb{N} \backslash V$, ou seja k é o menor natural para o qual não vale P, o que quer dizer que P vale para todos os naturais menores que k implicando $P(k)$ ser verdadeira, o que é um absurdo. Portanto $V = \mathbb{N}$. □

Agora temos condições de demonstrar o Teorema da Mudança de Base.

Teorema 2.3 (Mudança de Base)**.** *Dados a, $b \in \mathbb{N}$, com $b > 1$, existem números naturais c_0, c_1,...,c_n menores do que b, univocamente determinados, tais que $a = c_0 + c_1 b + c_2 b^2 + \ldots + c_n b^n$.*

Demonstração. A demonstração será por indução em a. Note que, se $a = 0$, basta tomar $c_i = 0$, $0 \leq i \leq n$. Nossa hipótese de indução é que todo natural menor do que a tem representação $d_0 + d_1 b + d_2 b^2 + \ldots + d_k b^k$ em que k e cada d_i são naturais e cada $d_i < b$ é único.

Aplicando a divisão euclidiana para a e b temos que $a = bq + r$ com $r < b$ com q e r únicos. Como $q < a$, logo $q = d_0 + d_1 b + d_2 b^2 + \ldots + d_k b^k$ donde segue que $a = r + d_0 b + d_1 b^2 + d_2 b^3 + \ldots + d_k b^{k+1}$. Daí basta tomar $n = k + 1$, $c_0 = r$ e $c_i = d_{i-1}$ com $1 \leq i \leq n$ e esta forma de representar a é única pois herda a unicidade dos números d_i e r. □

Os números c_0, $c_1 b$, c_2, $\ldots + d_n$ são ditos os "dígitos" de a na base b, comumente representados na forma $a = (c_n c_{n-1} \ldots c_0)_b$.

A demonstração do Teorema 2.3 acima nos fornece um algoritmo para passar um número a para a base b, basta aplicar a divisão euclidiana como segue:

$$\begin{aligned} a &= bq_0 + r_0, \quad r_0 < b, \\ q_0 &= bq_1 + r_1, \quad r_1 < b, \\ q_1 &= bq_2 + r_2, \quad r_2 < b, \\ &\vdots \end{aligned}$$

Como $a > q_0 > q_1 > \ldots > q_i$, $i \in \mathbb{N}$, em algum momento ter-se-á $q_{n-1} < b$ para algum n. Portanto, de

$$q_{n-1} = bq_n + r_n,$$

segue que $q_n = 0 \Longrightarrow 0 = q_j$ para todo $j \geq n$. Daí tem-se, $0 = r_k$ para todo $k \geq n + 1$. Finalmente, temos que

$$a = r_0 + r_1 b^1 + \ldots + r_n b^n$$

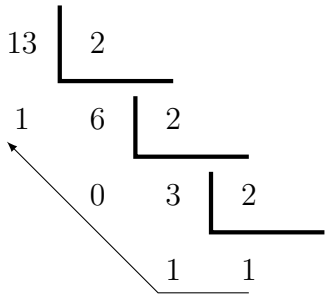
Como exemplo vamos passar o 13 para a base binária:

$$\begin{aligned} 13 &= 2 \cdot 6 + 1 \\ 6 &= 2 \cdot 3 + 0 \\ 3 &= 2 \cdot 1 + 1 \\ 1 &= 2 \cdot 0 + 1 \end{aligned}$$

Donde

$$\begin{aligned} 13 &= 2(2 \cdot 3 + 0) + 1 \\ 13 &= 2^2 \cdot 3 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ 13 &= 2^2 \cdot (2 \cdot 1 + 1) + 0 \cdot 2^1 + 1 \cdot 2^0 \\ 13 &= 2^3 \cdot 1 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ \therefore \quad 13 &= (1101)_2 \end{aligned}$$

Vejamos um método mais prático, ilustrado pela maneira tradicional de se representar a divisão, para converter da base 10 para base 2. Transformaremos 13 em binário



Portanto pelo método das divisões sucessivas
Temos que o 13 na base binária é 1101, representamos assim $13 = (1101)_2$.

novamente:

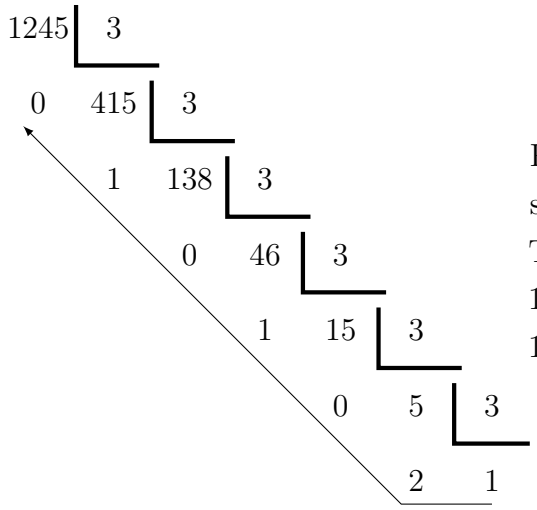
Para outro número qualquer, basta repertimos o processo descrito acima, até o quociente chegar ao valor 1, daí teremos a representação binária do referido número que será a concatenação do último quociente com os restos das divisões anteriores (indo do último até o primeiro, como indica a seta na figura acima). A tarefa agora é transformar esse algoritmo num programa em Python.

Observe que os quocientes vão mudando de valor, sempre a parte inteira da metade do valor anterior; isto acontece até o último quociente ter valor 1 (isso sugere o uso de um **while** [enquanto]). Em cada passo temos que armazenar os respectivos restos (isso sugere o uso de uma lista ou uma string vazia), onde o último valor dessa lista será o último quociente, depois basta imprimir a lista na ordem inversa. Vamos ao Programa 2.3:

Código 2.3 – Conversão binária

```
1  n =int(input('Digite o número a ser convertido pra binário: '))
2  base = " "
3  q = n
4  while n >= 2:
5      base = base + str(n%2)
6      n = n // 2
7      if n == 1:
8          base = base + str(n)
9  print('O %d na base 2 é '%q,base[::-1])
10 RESTART:
11 Digite o número a ser convertido pra binário: 13
12 O 13 na base 2 é  1101
```

Escolhemos a base 2 pois os computadores trabalham utilizando a base binária, mas podemos escrever um número natural n para qualquer base b ($b > 1$) como prova o Teorema 2.3. Por exemplo vamos passar o número $n = 1245$ para a base $b = 3$.



Portanto pelo método das divisões sucessivas
Temos que o 1245 na base 3 é 1201010, representamos assim $1245 = (1201010)_3$.

Vejamos como implementar um programa em Python que forneça a mudança de base, onde serão solicitados o número n e a base b para a qual se deseja fazer a conversão. A ideia é dividir o número n sucessivas vezes pela base b, bem semelhante ao que foi feito para a base 2.

Código 2.4 – Conversão para base b

```
b = int(input('Digite a base para a qual quer converter: '))
n = int(input('Digite o número que quer converter: '))
k = n
base=[]
while n >= b:
    base.append(n%b)
    n = n // b
if n <= b-1:
    base.append(n)
print('%d na base %d é'%(k,b),base[::-1])
RESTART:
Digite a base para a qual quer converter: 3
Digite o número que quer converter: 1245
1245 na base 3 é [1, 2, 0, 1, 0, 1, 0]
>>>
Digite a base para a qual quer converter: 17
Digite o número que quer converter: 810
810 na base 17 é [2, 13, 11]
```

No Programa 2.4 optou-se por apresentar a resposta em um vetor, para que quando a base for maior que 10, não haja confusão. Assim deve-se interpretar $810 = [2, 13, 11] = (2, 13, 11)_{17} = 2 \cdot 17^2 + 13 \cdot 17^1 + 11 \cdot 17^0$.

Claramente, dado um número $(\alpha_n \alpha_{n-1} \dots \alpha_0)_b$ numa base qualquer b, com $\alpha_i < b$

com $i \in \{0, 1, \ldots, n\}$ podemos convertê-lo para base 10 efetuando o cálculo:

$$\alpha_n b^n + \alpha_{n-1} b^{n-1} + \ldots + \alpha_0 b^0$$

Por exemplo, $(2014)_5 = [2, 0, 1, 4]_5$ é 259, pois $2 \cdot 5^3 + 0 \cdot 5^2 + 1 \cdot 5^1 + 4 \cdot 5^0 = 259$
O Programa 2.5 efetua tal conversão.

Código 2.5 – Conversão da base b para base 10

```
b=int(input("Forneca a base: "))
n = []
x,i = 1,0
a, N = 0, 0
while True:
    a = int(input("Digite o %d algarismo ou um valor >= %d para
   sair:"%(x,b)))
    if a >= b:
        break
  n.append(a)
  x += 1
while i <= len(n)-1:
  N += n[i]*b**((len(n)-1-i))
  i +=1
print("O número" ,n,'(base %d) é %d na base dez'%(b,N))
RESTART:
Forneca a base: 17
Digite o 1 algarismo ou um valor >= 17 para sair: 2
Digite o 2 algarismo ou um valor >= 17 para sair: 1
Digite o 3 algarismo ou um valor >= 17 para sair: 3
Digite o 4 algarismo ou um valor >= 17 para sair: 1
Digite o 5 algarismo ou um valor >= 17 para sair: 1
Digite o 6 algarismo ou um valor >= 17 para sair: 20
O número [2, 1, 3, 1, 1] (base 17) é 172840 na base dez
```

2.3 Um pouco mais de aritmética

Nesta seção abordaremos mais alguns teoremas e problemas relacionados aos números naturais, e como a linguagem de programação Python se apresenta como um bom recurso didático na compreensão e aplicação dos resultados que seguem.

Comecemos tratando do cálculo do máximo divisor comum entre dois números, um clássico dentre os algoritmos. Logo após, apresentamos alguns resultados sobre números primos.

2.3.1 Máximo Divisor Comum

Dados dois naturais a e b ambos não nulos, diremos que d é um divisor comum de a e b se $d|a$ e $d|b$.

Por exemplo, 3 é divisor comum de 15 e 27.

Definição 2.2. *Diremos que d é o máximo divisor comum de a e b se:*

i) *d é um divisor comum de a e b;*

ii) *Se c é um divisor comum de a e b, então $c|d$.*

Denotaremos o mdc de a e b por mdc(a,b).

Seguem alguns exemplos e propriedades. Nos itens que seguem a e b são naturais diferentes de zero.

- mdc(2,4)=2;
- mdc(1,a)=1 e mdc(a,a)=a;
- mdc(a,b)=$a \iff a|b$;
- mdc(a,b)=mdc(b,a).

O mdc(a,b) quando existe é único. De fato, seja d =mdc(a, b), suponhamos que exista outro d' =mdc(a,b) assim teríamos que $d \leq d'$ e $d' \leq d$, pelo Item **ii** da Definição 2.2. Conclui-se assim que $d = d'$.

Outro fato de extrema relevância é que sempre existe o mdc(a,b) entre dois naturais a e b não nulos. Para verificar tal propriedade, basta tomar o maior elemento comum nos conjuntos dos divisores de a e b; tal elemento é o mdc(a,b) e será no mínimo 1, pois 1 divide qualquer número.

Quando o mdc(a,b)=1, a e b são ditos primos entre si.

A seguir vejamos um método que mostra como encontrar o mdc(a,b). Tal método é conhecido por *Algoritmo de Euclides*. Mas antes precisamos de um resultado preliminar que será apresentado como um problema:

Problema 2.1. *Dados a,b, $n \in \mathbb{N}$ não nulos, com $a \leq na \leq b$, então mdc(a,b)=mdc(a,$b-na$)*

Solução: Seja d = mdc(a,b), pelo fato de $d|a$ e $d|b$ tem-se que $d|b-na$. Seja d' = mdc(a,$b-na$), tem-se que $d'|a$ e $d'|b-na$ logo $d'|b-na+na$, isto é, $d'|b$, portanto $d = d'$. ■

Dados a, $b \in \mathbb{N}$ não nulos, com $a \leq b$, o procedimento abaixo é o *Algoritmo de Euclides*. Tal algortimo é utilizado para se calcular o máximo divisor comum entre dois números.

Algoritmo de Euclides:

Caso: $a = 1$ ou $a = b$ ou $a|b$ teremos que mdc(a,b)=a.

Caso: $a \nmid b$ pela divisão euclidiana temos:

$$b = aq_1 + r_1 \qquad \text{com} \quad r_1 < a$$

Se $r_1|a$ temos que $r_1 = \text{mdc}(a,r_1) = \text{mdc}(a,b - aq_1) = \text{mdc}(a,b)$

E se $r_1 \nmid a$ pela divisão euclidiana temos:

$$a = r_1q_2 + r_2 \qquad \text{com} \quad r_2 < r_1$$

Se $r_2|r_1$ temos que $r_2 = \text{mdc}(r_1,r_2) = \text{mdc}(r_1,a - r_1q_2) = \text{mdc}(r_1,a) = \text{mdc}(b - aq_1,a) = \text{mdc}(b,a) = \text{mdc}(a,b)$

E se $r_2 \nmid r_1$, aplicamos a divisão euclidiana novamente

$$r_1 = r_2q_3 + r_3 \qquad \text{com} \quad r_3 < r_2.$$

Repetindo este procedimento, teremos uma sequência de restos decrescente, desta foram, em algum momento tal processo terá que parar devido ao Axioma 2.1, uma vez que o conjunto formado pelos restos dessas divisões é limitado inferiormente pelo zero. Então existirá algum $n \in \mathbb{N}$ com $r_n|r_{n-1}$, portanto temos que $r_n = \text{mdc}(a,b)$.

O *Algoritmo de Euclides* se apresenta de uma maneira mais prática através do dispositivo abaixo, conhecido como *jogo da velha*.

	q_1	
b	a	
r_1		

	q_1	q_2	
b	a	r_1	
r_1	r_2		

	q_1	q_2	$\ldots$	q_{n-1}	q_n	q_{n+1}
b	a	r_1	$\ldots$	r_{n-2}	r_{n-1}	$r_n = \text{mdc}(a,b)$
r_1	r_2	r_3	$\ldots$	r_n		

Exemplificando vamos calcular o mdc(255, 221)

	1	6	2
255	221	34	17
34	17		

Logo mdc(255, 221) = 17

Nosso objetivo agora é fazer um programa que calcule o mdc de dois números fornecidos pelo usuário.

Para isso, basta notar que o mdc entre os dois números é o resto r_n que divide o resto r_{n-1} para algum n. Convém notar que as divisões começam por b e a e depois a por r_1 e a assim sucessivamente, sugerindo o uso de um **while** até haver divisão exata.

Código 2.6 – mdc(a, b)

```
a = int(input("Digite o primeiro número: "))
b = int(input("Digite o segundo número: "))
x,y=a,b
while b % a != 0:
    a, b = b%a, a
print(" mdc(%d,%d) = %d" %(x,y,a))
RESTART:
Digite o primeiro número: 255
Digite o segundo número: 221
mdc(255,221) = 17
```

Apenas para ilustrar, segue o código abaixo com a função mdc

Código 2.7 – Função mdc(a, b)

```
def mdc(a,b):
    while b%a!=0:
        b,a=a,b%a
    return a
>>> # O Python já reconheceu a nova função mdc
>>> # basta chamá-la, como segue os exemplos
>>> mdc(255,221)
17
>>> mdc(54,16)
2
```

2.3.2 Primos

Antes do primeiro problema, vamos definir número primo.

Definição 2.3. *Um número natural $p > 1$ é dito primo se é divisível apenas por 1 e por si mesmo.*

Observação: Quando n não for primo, dizemos que n é composto.

Problema 2.2. *Dado $n \in \mathbb{N}$ com $n > 1$ como decidir se n é primo ou composto?*

Solução: Caso $n = 2$ será primo por definição, caso contrário basta dividir n por cada $d \in \{2, 3, \ldots, n-1\}$. Se ocorrer de alguma divisão ser exata para algum d, isto é,

deixar resto zero pode-se parar o processo pois n será composto apresentando no mínimo três divisores: 1, d e n. Caso contrário n será primo. ■

Agora vamos fazer um programa que nos forneça a informação se dado $n > 1$ natural é primo ou não. Observe que durante a solução apresentada temos dois conceitos lógicos que são traduzidos por duas palavras reservadas do Python (**for** e **if**). O uso do **for** de dá, pois vamos fazer divisões partindo do 2 até o antecessor do número fornecido, enquanto que o **if** se mostra útil, pois analisamos inicialmente se o numero fornecido é 2 e depois dentro do **for** analisamos os restos das divisões. Isto posto vejamos o programa abaixo:

Código 2.8 – Teste de primalidade

```
n=int(input('Digite um número natural maior que 1: '))
if n==2:
    print('%d é primo'%n)
else:
    for d in range(2,n):
        if n%d==0 and d!=n-1:
            print('%d não é primo'%n)
            break
        else:
            if d==n-1:
            print('%d é primo'%n)
RESTART:
Digite um número natural maior que 1: 7493
7493 não é primo
>>>
RESTART:
Digite um número natural maior que 1: 18446744073709551617
18446744073709551617 não é primo
```

Sabemos que dado $n \in \mathbb{N}$, temos que n é primo ou n é composto. Um fato de extrema relevância é que caso n seja composto, existem primos que quando multiplicados reproduzem n, ou seja, existem primos que compõem n. Tal afirmação é o conteúdo do próximo teorema.

Teorema 2.4 (Teorema Fundamental da Aritmética)**.** *Seja $n \in \mathbb{N}$, $n > 1$. Então n se escreve de modo único (a menos da ordem dos fatores) como produto de primos.*

Demonstração. Usaremos o Segundo Princípio de Indução sobre n. O resultado é válido para $n = 2$. Suponha o resultado válido para cada natural $2 \leq k < n$. Se n for primo o resultado é claramente válido. Caso n seja composto, existem n_1 e n_2, ambos menores

que, n com $n = n_1 n_2$. Pela hipótese de indução $n_1 = p_1 p_2 \dots p_r$ e $n_2 = q_1 q_2 \dots q_s$ logo $n = p_1 \dots p_r q_1 \dots q_s$.

Agora vamos provar a unicidade. Suponha que $n = p_1 p_2 \dots p_r = q_1 q_2 \dots q_s$ com p_i, $1 \leq i \leq r$, e q_j, $1 \leq j \leq s$, primos. Temos então que algum $p_i | q_1 q_2 \dots q_s$, o que implica $p_i = q_j$; reordenando os primos na decomposição de n podemos supor que sejam $p_1 = q_1$. Assim teremos $p_2 \dots p_n = q_2 \dots q_s$ assim teremos, por hipótese de indução, que $r = s$ uma vez que $q_2 \dots q_s < n$. Portanto teremos $p_i = q_j$, $2 \leq i, j \leq r$. □

Para exemplificar, vejamos a decomposição em fatores primos de alguns número:

$6 = 2 \cdot 3 = 2 \times 3$; $15 = 3 \cdot 5$; $18 = 2 \cdot 3 \cdot 3 = 2 \cdot 3^2$.

Uma maneria prática na qual um número é decomposto se apresenta no método abaixo:

n	p_1
n_1	p_2
n_2	p_3
$\vdots$	$\vdots$
n_{r-1}	p_r
1	

6	2
3	3
1	

15	3
5	5
1	

18	2
9	3
3	3
1	

Se $n = n_0$, cada n_i é o quociente de n_{i-1} por p_i com $i = 1, 2, \dots, r$, e o número p_i corresponde ao menor primo que divide n_{i-1}. Tal processo acaba quando o quociente da divisão de n_{r-1} por p_r é 1. Assim, a forma fatorada de n em primos será $n = p_1 p_2 \dots p_r$.

Observação: Forma fatorada e decomposição em primos são sinônimos.

Esse método por si só já estrutura um programa que, dado n, nos retorne a forma fatorada de n.

Quando tem-se fatores repetidos na decomposição de n, diremos que tal fator tem multiplicidade igual ao número de vezes em que aparece na fatoração. Por exemplo em $18 = 2 \cdot 3 \cdot 3$ dizemos que 3 tem multiplicidade 2.

Assim a fatoração de n de forma geral é $n = p_1^{r_1} p_2^{r_2} \dots p_k^{r_k}$ com $r_i > 0$ com $i = 1, \dots, k$.

O Programa 2.9 exibe os fatores (e suas multiplicidades) de um número n em um vetor.

Código 2.9 – Decomposição de n

```
1 n = int(input('Entre com o número: '))
2 fatores = []
3 d = 2
4 while n > 1:
```

```
5     if n%d == 0:
6         n = n/d
7         fatores.append(d)
8     else:
9         d += 1
10 print (fatores) # esse é o vetor com todos os fatores de n
11 RESTART:
12 Entre com o número: 18
13 [2, 3, 3]
14 >>>
15 RESTART:
16 Entre com o número: 7493
17 [59, 127]
```

De posse do Teorema 2.4 vamos apresentar outro teste de primalidade devido a Eratóstenes[3]. Tal método é melhor que o mostrado no Problema 2.2, no entanto também é lento. Outros testes de primalidade podem ser encontrados em (MOREIRA; SALDANHA, 2011).

Teorema 2.5. *Seja $n \in \mathbb{N}$, com $n > 1$. Se nenhum p primo com $p \leq \sqrt{n}$ divide n, então n é primo.*

Demonstração. Iremos demonstrar utilizando a contra-positiva, ou seja, mostraremos que se n é composto então existe p primo com $p \leq \sqrt{n}$ que divide n. Seja n um número composto. Suponha, por contradição, que t é o menor primo que divide n e $t > \sqrt{n}$. Pelo Teorema Fundamental da Aritmética 2.4 temos que ter, no mínimo, outro primo t_1 que divide n. Então $t_1 \geq t > \sqrt{n}$, de onde, $n = t \cdot t_1 > n$, absurdo. □

Exemplificando, considere $n = 71$. Como $\lfloor\sqrt{71}\rfloor = 8$, em que $\lfloor x \rfloor$ é o maior inteiro menor do que ou igual a x, com $x \in \mathbb{R}$, para testar se 71 é primo, basta dividir este número apenas pelos primos entre 2 e 8.

O Programa 2.10, mostra este teste de primalidade. Neste programa fizemos as divisões de n por números de 2 até $\lfloor\sqrt{n}\rfloor$, usando o **while** e **if**.

Código 2.10 – Teste de primalidade (Eratóstenes)

```
1 n = int(input("Digite um número > 1, que vamos testar se é primo:
    "))
2 p = int(n**0.5)
3 while p >= 1: # Temos que testar p=1, por causa das raízes de 2 e
    3
```

[3] Matemático grego que viveu por volta dos anos 230 a.C

```
4       if n%p == 0 and p !=1:
5           print(' O número %d não é primo.' %n)
6           break
7       if p == 1:
8           print('O número %d é primo.' %n)
9       p = p - 1
10  RESTART:
11  Digite um número > 1, que vamos testar se é primo: 4294967297
12  O número 4294967297 não é primo.
```

O Programa 2.11 é uma adaptação do Programa 2.10, no qual foi criada uma função que retorna **True** se um dado n natural é primo e **False**, caso contrário.

Código 2.11 – Função ehprimo

```
1   def ehprimo(n):
2       p = int(n**0.5)
3       while p >= 1:
4           if n%p == 0 and p !=1:
5               return False
6               break
7           if p == 1:
8               return True
9           p = p - 1
10  RESTART:
11  >>> ehprimo(111)
12  False
13  >>> ehprimo(65537)
14  True
```

De modo geral não é simples determinar a primalidade ou não de um número natural grande sem ferramentas computacionais. Podemos ilustar tal afirmação resgatando um episódio envolvendo dois grandes matemáticos, que são Fermat[4] e Euler[5], conforme apresentado em (LEITE, 1985; HEFEZ, 2011; PENHA, 1983).

Fermat em uma carta enviada a seu amigo Bernard Frenide de Bessy em 1640 afirmava crer que os números da forma $F_n = 2^{2^n} + 1$ (números de Fermat ou primos de Fermat) fossem primos para todo natural n.

Vejamos alguns valores de F_n:

$$F_0 = 2^{2^0}+1 = 2+1 = 3, \quad F_1 = 2^{2^1}+1 = 4+1 = 5, \quad F_3 = 2^{2^3}+1 = 256+1 = 257$$

[4] Pierre de Fermat (1601-1665) matemático francês do séc. XVII; deu várias contribuições em Teoria dos Números

[5] Leonhard Euler (1707-1783) matemático suíço, figura entre os maiores e mais produtivos matemáticos de todos os tempos.

$F_4 = 2^{2^4} + 1 = 65537$ que são de fato primos.

Já $F_5 = 2^{2^5} + 1 = 4294967297$ não foi verificado ser primo por Fermat, provavelmente por ser muito grande. No entanto Euler, quase 100 anos depois, em 1732 mostrou que F_5 é composto com um fator igual a 641. De fato tem-se que $F_5 = 4294967297 = 641 \times 6700417$. Maiores detalhes de como Euler conseguiu fatorar F_5 podem ser encontrado em em (PENHA, 1983).

Podemos verificar a não primalidade de F_5 e sua fatoração utilizando os Códigos 2.10 e 2.11 respectivamente.

A fatoração de um dado n, nos possibilita determinar, entre outras coisas, as citadas abaixo relativas a n :

1. Quantos divisores n possui;
2. Quais são tais divisores, qual é a soma e o produto destes divisores.

De modo geral seja $n = p_1^{r_1} p_2^{r_2} \dots p_k^{r_k}$ e sejam D, $D(n)$, $S(n)$ e $P(n)$ o número de divisores, o conjunto dos divisores, a soma dos divisores e o produto do divisores de n. A título de ilustraçao, D e $S(n)$ podem ser calculados pelas fórmulas:

$$D = (r_1 + 1)(r_2 + 1) \dots (r_k + 1)$$

$$S(n) = \frac{p_1^{r_1+1} - 1}{p_1 - 1} \dots \frac{p_k^{r_k+1} - 1}{p_k - 1}$$

$$P(n) = n^{\frac{D}{2}}$$

A demonstração da expressão para D pode ser consultada em (LIMA et al., 2006), enquanto que para ver uma demonstração sobre $S(n)$ consultar (HEFEZ, 2011). Provemos a fórmula para $P(n)$ a seguir.

Problema 2.3. *Dado $n \in \mathbb{N}$, com $n > 1$, prove que o produto dos divisores de n, denotado por $P(n)$, é igual a $n^{\frac{D}{2}}$, onde D é o número de divisores de n.*

Solução: Seja $\{d_1, d_2, \dots, d_D\}$ o conjunto dos divisores de n. Temos então que $n = d_i \cdot d_j$ com $i \neq j$ para algum $i, j \in \{1, \dots, D\}$, segue que $P(n) = d_1 \cdot d_2 \cdot \ldots \cdot d_D$ pode ser reordenado de tal modo que a cada dois fatores o resultado seja sempre n, logo $P(n) = \underbrace{n \cdot n \cdot \ldots \cdot n}_{\frac{D}{2} \text{ vezes}} = n^{\frac{D}{2}}$.

Caso $n = k^2$ teremos para algum i e j, $d_i = d_j = k$ que é contado duas vezes no produto e no entanto só uma vez no conjunto dos divisores. ■

Como consequência os únicos números que tem uma quantidade ímpar de divisores são os quadrados perfeitos.

A ideia agora é elaborar um programa que dado $n \in \mathbb{N}$, $n > 1$ calculemos D, $D(n)$, $S(n)$ e $P(n)$ mencionados acima.

A estratégia se baseia no fato de que para cada divisor d_i de n temos $n = d_i \cdot d_i'$, ou seja os divisores aparecem aos pares. Então criaremos um vetor e armazenaremos os d_i e seus respectivo d_i' e faremos todos os testes de divisibilidade enquanto $d \leq \sqrt{n}$. De posse do vetor dos divisores basta varrer todos seus elementos com dois acumuladores, um pra soma e outro para o produto. O número de divisores é simplesmente **len**(vetor do divisores). O Código 2.12 apresenta uma possível solução em Python.

Código 2.12 – Quantidade, conjunto, soma e produto dos divisores de n

```
n=int(input('Entre com o valor de n: '))
d,S,P=1,0,1
vetor=[]
while d <= n**0.5:
    if n%d==0:
        vetor.append(d)
        vetor.append(n//d)
    d=d+1
divisores=list(set(vetor)) # eliminando repetições
divisores.sort() # deixando os elementos na ordem crescente
for i in range(len(divisores)):
    S=S+divisores[i]
    P=P*divisores[i]
print('O número  de divisores de %d é D=%d '%(n,len(divisores)))
print('D(%d) ='%n,divisores)
print('A soma de todos os divisores de %d é S(%d)=%d'%(n,n,S))
print('E o produto de todos os divisores de %d é
   P(%d)=%d'%(n,n,P))
RESTART:
Entre com o valor de n: 60
O número  de divisores de 60 é D=12
D(60) = [1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60]
A soma de todos os divisores de 60 é S(60)=168
E o produto de todos os divisores de 60 é P(60)=46656000000
```

No embalo dos primos temos mais dois problemas que surgem naturalmente, são eles:

1. Quantos números primos existem?

2. Dado $n \in \mathbb{N}$ quais são todos os primos até n?

A resposta aos questionamentos acima, seguem no próximo teorema e no próximo problema, devido a Euclides e Eratóstenes respectivamente.

Teorema 2.6 (Euclides)**.** *Existem infinitos números primos.*

Demonstração. Suponhamos que existam apenas um número finito de números primos $p_1, p_2, \ldots, p_k$ e seja $n = p_1 p_2 \ldots p_k + 1$. Pelo Teorema 2.5 um dos p_i, $i = 1, \ldots k$, divide n; ora mas como $p_i | p_1 p_2 \ldots p_i \ldots p_k$ temos que $p_i | n - p_1 p_2 \ldots p_k = 1$, o que é um absurdo. □

Problema 2.4 (Crivo de Eratóstenes)**.** *Dado $n \in \mathbb{N}$ com $n > 1$, determine todos os primos até n (inclusive).*

Solução: A ideia é montar uma tabela com todos os números naturais de 2 até n e fazer o seguinte procedimento:

1. deixar o 2 e riscar todos os seus múltiplos até n;
2. deixar o próximo número primo maior que 2, que é 3, e riscar todos os seus múltiplos até n;
3. deixar o próximo número primo maior do que 3, que é 5 e riscar todos os seus múltiplos até n;

   E assim sucessivamente até não ser mais possível tal procedimento. Os números que ficarem intactos (não forem riscados) serão os primos até n.

Para efeitos ilustrativos, vejamos o processo com $n = 50$

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~

Logo os primos até 50 são $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47\}$ ■

Vejamos agora um programa que nos retorna todos os primos até um dado n fornecido pelo usuário. Já percebemos que existem diversas formas de resolver o mesmo

problema, também não é diferente com os programas, vamos utilizar a função ehprimo do código 2.10 e a partir daí testar a função *ehprimo*, do Programa 2.11, dentro de um **for** e os números que forem primos serão armazenados em um vetor; ao fim do laço do **for** basta imprimir tal vetor para visualizaramos os primos até n.O Código 2.13 faz este trabalho.

Código 2.13 – Crivo de Eratóstenes

```
def ehprimo(n):
    p=int(n**0.5)
    while p>=1:
        if n%p==0 and p!=1:
            break
        if p==1:
            return True
        p=p-1
vetor_primos=[]
n=int(input('Digite o número de parada: '))
for i in range(2,n+1):
    if ehprimo(i):
        vetor_primos.append(i)
print('Todos os primos até %d são'%n,vetor_primos)
RESTART:
Digite o número de parada: 50
Todos os primos até 50 são [2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
   31, 37, 41, 43, 47]
```

Para encerrarmos este capítulo, vejamos alguns números especiais: primos gêmeos, números perfeitos e primos de Mersene[6].

Definição 2.4. *Dizemos que dois primos p e q são gêmeos se $p - q = 2$.*

Vejamos abaixo um programa que mostra todos os pares de primos gêmeos até um dado n fornecido pelo usuário.

Código 2.14 – Primos gêmeos

```
primos=[]
gêmeos=[]
for i in range(2,int(input("n: "))+1):
    if all([i%a!=0 for a in range(2,int(i**0.5)+1)]):
        primos.append(i)
for i in range(len(primos)):
    if primos[i]+2 in primos:
```

[6] Marin Mersenne (1588-1648) matemático francês cujo nome é lembrado pelos primos da forma $2^p - 1$

```
8         gêmeos.append([primos[i],primos[i]+2])
9 print('Os pares de primos gêmeos são: ',gêmeos)
10 RESTART:
11 n: 50
12 Os pares de primos gêmeos são:  [[3, 5], [5, 7], [11, 13], [17,
    19], [29, 31], [41, 43]]
```

Podemos encontrar em (WATANABE, 2013) um resultado muito interessante sobre primos gêmeos devido Yitang Zhang[7]. Em (WATANABE, 2013) também encontra-se o maior par de primos gêmeos descoberto até a data de sua publicação. A saber, $3576801695685 \times 2^{666669} - 1$ e $3576801695685 \times 2^{666669} + 1$.

Atualmente, de acordo com o (The Prime Pages , 2018) o maior par de primos gêmeos é

$$2996863034895 \cdot 2^{1290000} - 1 \quad \text{e} \quad 2996863034895 \cdot 2^{1290000} + 1$$

Um número é chamado de perfeito quando a soma de seus divisores é igual ao seu dobro ou de forma equivalente se a soma de seus divisores próprios é igual a ele mesmo. (divisores próprios são os divisores de um número exceto ele mesmo)

Definição 2.5. *Seja n um natural e $S(n)$ a soma de seus divisores, se $S(n) = 2n$, dizemos que n é perfeito.*

Vejamos alguns exemplos de números perfeitos: 6, 28, 496 de fato o são, pois sejam $D(6)$, $D(28)$ e $D(496)$ o conjunto dos divisores próprios de cada número respectivamente, temos que:

$D(6) = \{1, 2, 3\}$ e $6 = 1 + 2 + 3$

$D(28) = \{1,\ 2,\ 4,\ 7,\ 14\}$ e $28 = 1 + 2 + 3 + 4 + 7 + 14$

$D(496) = \{1,\ 2,\ 4,\ 8,\ 16,\ 31,\ 62,\ 124,\ 248\}$ e

$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$

Finalmente tratemos os números de Mersenne. Os números da forma $M_p = 2^p - 1$ com p primo são chamados de números de Mersenne.

Definição 2.6. *Os primos de Mersenne são os primos da forma $M_p = 2^p - 1$, em que p é primo.*

Durante muitos séculos acreditou-se que os números de Mersenne fossem primos, contudo, Hudalricus Regius mostrou que $2^{11} - 1$ é composto.

[7] Matemático Chinês, professor da Universidade de New Hampshire

$M_2 = 2^2 - 1 = 3, \quad M_3 = 2^3 - 1 = 7, \quad M_5 = 2^5 - 1 = 31, \quad M_7 = 2^7 - 1 = 127$ são os primeiros primos de Mersenne. Para uma lista com todos conhecidos até a atualidade veja (Mersenne Researchs , 2018, 2018)

Existe um resultado que estabelece a relação entre números perfeitos e primos de Mersenne. Este é apresentado no próximo teorema, cuja demonstração pode ser encontrada em (HEFEZ, 2011).

Teorema 2.7. *Um número $n \in \mathbb{N}$ é perfeito par se, e somente se, $n = 2^{p-1}(2^p - 1)$ com $2^p - 1$ um primo de Mersenne.*

Observemos que:

$6 = 2^1(2^2-1)$, $28 = 2^2(2^3-1)$ e $496 = 2^4(2^5-1)$ são os números pefeitos associados aos Primos de Mersenne M_2, M_3 e M_5 citados acima.

Uma curiosidade: Segundo (Mersenne Researchs , 2018) o maior Primo de Mersenne é $2^{77232917} - 1$ com 23249425 digítos. Tal número foi descoberto por Pace, Woltman, Kurowski, Blosser & GIMPS em 26 de Dezembro de 2017; este é também o maior primo já descoberto e o número perfeito associado a ele é $2^{77232916} \cdot (2^{77232917} - 1)$.

Por fim apresentamos um programa que dado uma lista com os primeiros p números primos, retorna os primos de Mersenne $2^p - 1$.

Código 2.15 – Primos de Mersenne

```
def ehprimo(n):
p = int(n**0.5)
    while p >= 1:
        if n%p == 0 and p !=1:
            return False
            break
        if p == 1:
            return True
        p = p - 1
primos=[]
n=int(input('n: '))
for i in range(2,n+1):
    if ehprimo(i):
        primos.append(i)
mersenne=[]
for i in primos:
    if ehprimo(2**i-1):
        mersenne.append("2^%d - 1"%i)
print(mersenne)
```

```
20 RESTART:
21 n: 20
22 ['2^2 - 1', '2^3 - 1', '2^5 - 1', '2^7 - 1', '2^13 - 1', '2^17 -
     1', '2^19 - 1']
23
```

O programa acima retornou os seguintes primos de Mersenne $2^2 - 1$, $2^3 - 1$, $2^5 - 1$, $2^7 - 1$, $2^{13} - 1$, $2^{17} - 1$ e $2^{19} - 1$.

Finalizemos o capítulo salientando que os programas aqui apresentados podem ser melhorados e por vezes não apresentam uma melhor performace quando se considera tempo de execução e espaço de armazenamento. Contudo, a principal finalidade deste trabalho é fazer uma ponte entre o ensino de introdução à programação com a linguagem Python e o ensino de noções básicas da Teoria dos números.

Capítulo 3

Considerações finais

A presente dissertação teve como objetivo central apresentar a possibilidade da linguagem de programação Python como ferramenta para o ensino de aritmética. Neste sentido, apresentamos algumas funcionalidades do Python, aquelas que seriam utilizadas nos programas desenvolvidos, discutimos alguns resultados centrais de aritmética e propusemos programas em Python para tais resultados, buscando utilizar as demonstrações apresentadas para elaborar nossos códigos. Entendemos, que tais programas podem não ser os mais eficientes no que diz respeito a tempo de execução e espaço de armazenamento de dados, contudo, a ideia era propor um material que permitisse interligar o ensino de uma linguagem de programação com o entendimento dos resultados e de suas provas. Todos os programas foram desenvolvidos pelo autor desta dissertação. Vale ressaltar que, além de construir um programa é importante também vê-lo sendo executado linha a linha para entender seu funcionamento. Sugerimos consultar (Philip Guo, 2018) para tal ferramenta. A visualização da execução de um programa, além de facilitar o entendimento das estruturas básicas (**if**, **while** e **for**), estimula a busca por códigos mais enxutos.

O autor desta dissertação é professor no Instituto Federal de Goiás (IFG), Câmpus Itumbiara. Pretende-se, como trabalho futuro, adaptar esta dissertação em um projeto de ensino voltado para estudantes do nível médio. O IFG de Itumbiara contém uma infraestrutura favorável à execucão do projeto, pois conta com quatro laboratórios de informática e internet em todas salas. Assim, acredita-se que este projeto irá complementar a formação dos estudantes apresentando-lhes uma linguagem de programação moderna e tópicos de Teoria dos números, assunto praticamete esquecido no ensino médio.

Ressalta-se ainda que o autor pretende incrementar o material já proposto nesta dissertação, focando na construção de códigos acerca de outros assuntos em Matemática, tais como, matrizes, gráficos, trigonometria, funções elementares e estatística. Além disso, pretende-se trabalhar com os alunos mais sobre a linguagem Python: suas bibliotecas, jogos, orientação à objeto, banco de dados, dentre outros assuntos.

Referências

HEFEZ, A. **Elementos de Aritmética, 2ª Edição, Rio de Janeiro**. [S.l.]: SBM, 2011.

LEITE, P. F. Números de fermat. **Revista do Professor de Matemática**, v. 07, 1985.

LIMA, E. L. et al. A matemática do ensino médio, vol. 2. **Coleçao do Professor de Matemática, SBM**, 2006.

Masanori. **Python Zumbi 0**. 2018. Disponível em: <https://www.youtube.com/watch?v=6La690qlH5w&list=PLUukMN0DTKCtbzhbYe2jdF4cr8MOWClXc>. Acessado em 07/09/2018.

MENEZES, N. N. C. **Introdução à programação com Python–2ª edição: Algoritmos e lógica de programação para iniciantes**. [S.l.]: Novatec Editora, 2016.

Mersenne Researchs . **Prime numbers records**. 2018. Disponível em: <https://www.mersenne.org/primes/>. Acessado em 06/09/2018.

MORAN, J.; MASETO, M. T.; BEHRENS, M. A. **Novas Tecnologias E Mediação Pedagógica**. [S.l.]: Papirus, 2013. (Coleção Papirus Educação). ISBN 9788530805944.

MOREIRA, C. G. T. d. A.; SALDANHA, N. C. **Testes de primalidade: probabilísticos e determinísticos**. [S.l.: s.n.], 2011.

PENHA, G. de la. Euler e a teoria dos números. **Revista do Professor de Matemática**, v. 04, 1983.

Philip Guo. **Python tutor**. 2018. Disponível em: <http://pythontutor.com/>. Acessado em 15/01/2018.

Python Brasil. **Empresas Python**. 2018. Disponível em: <https://python.org.br/empresas/>. Acessado em 07/09/2018.

Python ORG. **Interative Python Online**. 2018. Disponível em: <https://www.python.org/shell/>. Acessado em 05/01/2018.

_____. **Python Documents**. 2018. Disponível em: <https://docs.python.org/3/library/math.html>. Acessado em 07/09/2018.

_____. **Python Download**. 2018. Disponível em: <https://www.python.org/>. Acessado em 12/01/2018.

SIDKI, S. **Introdução à teoria dos números**. [S.l.]: IMPA, 1975.

SILVA, H. A. d. Regimento do profmat - sbm. 2016.

SUMMERFIELD, M. **Programação em Python 3: Uma introdução completa à linguagem Python**. [S.l.]: Alta Books, 2012. (Biblioteca do Programador).

The Prime Pages . **Prime numbers**. 2018. Disponível em: <https://primes.utm.edu/largest.html#Mersenne>. Acessado em 05/09/2018.

WATANABE, R. Notícias espetaculares. **Revista do Professor de Matemática**, v. 82, p. 2–6, 2013.