Capítulo 2

Variáveis, expressões e declarações

2.1 Valores e tipos

Um *valor* é uma das coisas mais básicas trabalhadas por um programa, como uma letra ou um número. Os valores que vimos até aqui foram 1, 2, e "Alô, Mundo!"

Esses valores pertencem a diferentes *tipos*: 2 é um inteiro, e "Alô, Mundo!" é uma *string*, assim chamada pois contém uma "string" de letras. Você (e o interpretador) podem identificar string porque elas são demarcadas com aspas.

A sentença `print` também funciona para inteiros. Usamos o comando `python` para inicializar o interpretador.

```
python
>>> print(4)
4
```

Se você não está tão certo sobre qual é o tipo de um valor, o interpretador pode te falar.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Sem muita surpresa, strings são do tipo `str` e inteiros são do tipo `int`. Um pouco menos óbvio, números com casas decimais são do tipo `float`, porque esses números são representados em um formato chamado *ponto flutuante*.

class!float

```
>>> type(3.2)
<class 'float'>
```

Mas o que dizer sobre valores como "17" e "3.2"? Eles parecem números, mas estão demarcados com aspas assim como as strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Eles são strings.

Quando você digita um inteiro muito grande, você deve estar tentado a usar uma vírgula entre grupos de três dígitos 1,000,000. Isso não é um inteiro válido em Python, mas é válido:

```
>>> print(1,000,000)
1 0 0
```

Bom, não é o que esperávamos! Python interpreta 1,000,000 como uma sequência de inteiros separada por vírgulas, a qual é mostrada com um espaço entre cada inteiro.

index{erro semântico}

Esse é o primeiro exemplo em que vemos um erro semântico: o código é executado sem mostrar nenhuma mensagem de erro, mas não faz a coisa "certa".

2.2 Variáveis

Um dos recursos mais poderosos da linguagem de programação é a capacidade de manipular *variáveis*. Uma variável é um nome que faz referência a um valor.

Uma *declaração por atribuição* cria novas variáveis e atribui valores a elas:

```
>>> message = 'E agora, para algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897931
```

Nesse exemplo, foram feitas três atribuições. A primeira designa uma string para uma nova variável chamada `mensagem`; a segunda atribui o valor inteiro 17 para `n`; a terceira atribui o valor (aproximado) de π para a variável `pi`.

Para mostrar o valor de uma variável, você pode usar um comando print:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

O tipo de uma variável corresponde ao tipo do valor ao qual ela se refere.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Nomes de variáveis e palavras-chave

Os programadores geralmente escolhem nomes para suas variáveis que são significativos e documentam para que a variável vai ser usada.

Nomes de variáveis podem ser arbitrariamente longos. Elas podem conter letras e números, mas elas não podem começar com um número. É válido usar letras maiúsculas, mas é uma boa ideia começar nomes de variáveis com uma letra minúscula (você verá o porquê mais tarde).

O caractere de sublinhado (_) pode aparecer em um nome. É frequentemente usado em nomes com várias palavras, como `meu_nome` ou `velocidade_de_uma_andorinha_sem_carga`. Nomes de variáveis podem começar com um caractere sublinhado, mas é bom evitar fazer isso a menos que se esteja escrevendo um código de biblioteca para outros.

Se você fornecer uma variável com nome inválido, você obter um erro de sintaxe:

```
>>> 76trombones = 'grande desfile'
SyntaxError: invalid syntax
>>> mais@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` é inválido porque começa com um número. `mais@` é inválido porque contém um caractere irregular, @. Mas o que há de errado com `class`?

Acontece que `class` é uma das *palavras-chave* do Python. O interpretador usa palavras-chave para reconhecer a estrutura do programa, e eles não podem ser usados como nomes de variáveis.

Python reserva 33 palavras-chave para o seu uso:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

Talvez você queira manter esta lista à mão. Se o interpretador reclamar sobre um de seus nomes de variáveis e você não sabe por quê, veja se ele está nesta lista.

2.4 Declarações

Uma *declaração* é uma parte do código que o Python interpreta e pode executar. Nós temos visto dois tipos de declarações: print como uma declaração ou uma atribuição.

Quando você digita uma declaração no modo interativo, o interpretador a executa e exibe os resultados, se houver um.

Um script geralmente contém uma sequência de declarações. Se houver mais de uma declaração, os resultados aparecerão à medida que as instruções são executadas.

Por exemplo, o script

```
print(1)
x = 2
print(x)
```

produz a saída

```
1
2
```

A declaração por atribuição não produz saída.

2.5 Operadores e operandos

Operadores são símbolos especiais que representam operações como adição e multiplicação. Os valores em que o operador é aplicado são chamados *operandos*.

Os operadores `+`, `-`, `*`, `/`, e `**` realizam respectivamente adição, subtração, multiplicação, divisão, e exponenciação, como nos seguintes exemplos:

```
20+32
hora-1
hora*60+minuto
minuto/60
5**2
(5+9)*(15-7)
```

Houve uma mudança no operador de divisão entre Python 2.x e Python 3.x. Em Python 3.x, a resposta dessa divisão é um resultado com ponto flutuante:

```
>>> minuto = 59
>>> minuto/60
0.9833333333333333
```

O operador de divisão em Python 2.x dividiria dois inteiros e truncaria o resultado para um inteiro:

```
>>> minute = 59
>>> minute/60
0
```

Para obter a mesma resposta em Python 3.x use a divisão inteira(//inteiro).

```
>>> minute = 59
>>> minute//60
0
```

Em Python 3.x a divisão de números inteiros funciona de maneira muito mais semelhante ao que você esperaria se colocasse a expressão em uma calculadora.

2.6 Expressões

Uma *expressão* é uma combinação de valores, variáveis e operadores. Um valor, por si só, é considerado uma expressão, assim como uma variável. O que será apresentado a seguir são expressões válidas (assumindo que tenha sido atribuído um valor à variável `x`)

```
17
x
x + 17
```

Se você digitar uma expressão no modo interativo, o interpretador a *avaliará* e mostrará o resultado:

```
>>> 1 + 1
2
```

Mas em um script, uma expressão, por si só, não faz nada! Isto geralmente causa muita confusão com iniciantes.

Exercício 1: Digite as expressões a seguir no interpretador Python para ver o que elas fazem:

```
5
x = 5
x + 1
```

2.7 Ordem das operações

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das *regras de precedência*. Para operadores matemáticos, o Python segue a convenção matemática. A sigla *PEMDAS* é uma maneira útil de lembrar as seguintes regras:

- *P*arênteses têm a precedência mais alta e podem ser utilizados para que forçar uma expressão a ser avaliada na ordem desejada. Como as expressões em parênteses são avaliadas primeiro, `2 * (3-1)` é 4, e `(1+1)**(5-2)`é 8. Você também pode usar parênteses para tornar uma expressão mais fácil de ser lida, como em `(minutos * 100) / 60`, mesmo que isto não resulte em uma mudança no resultado final.

- *E*xponenciação tem a segunda precedência mais alta, então `2**1+1` é 3, não 4, e `3*1**3` é 3, não 27.

- *M*ultiplicação e *D*ivisão possuem a mesma precedência, que é maior que a *A*dição e *S*ubtração, que também têm a mesma precedência. Então `2*3-1` é 5, não 4, e `6+4/2` é 8, não 5.

- Operadores com a mesma precedência são avaliados da esquerda para direita. Desta maneira, a expressão `5-3-1` é 1, não 3, pois a opreção `5-3zzzz` acontece primeiro e só posteriormente `1` é subtraído do `2`.

Quando estiver em dúvida sobre como o computador vai interpretar uma operação, sempre coloque os parênteses nas suas expressões para garantir que os cálculos serão executados na ordem que você pretende.

2.8 Operador de módulo

O *operador de módulo* funciona em inteiros e produz o restante quando o primeiro operando é dividido pelo segundo. Em Python, o operador módulos é um sinal de porcentagem `%`. a sintaxe é a mesma que para outros operadores:

```
>>> quociente = 7 // 3
>>> print(quociente)
2
>>> resto = 7 % 3
>>> print(resto)
1
```

Assim 7 dividido por 3 é 2 com resto 1 .

O operador de módulo acaba por ser surpreendentemente útil. Para exemplo, você pode verificar se um número é divisível por outro: se `x \% y` é zero, então `x` é divisível por `y`.

Você também pode extrair o dígito ou os dígitos mais à direita de um número. Por exemplo, `x \% 10` retorna o dígito mais à direita de `x` (na base 10). Similar, `x \% 100` retorna os dois últimos dígitos.

2.9 Operações com String

O operador `+` possui uma função ao ser colocado com operandos do tipo string, porém esta função não é aditiva, do modo em que estamos acostumados na

matemática. Ao invés de somar, ele executa uma *concatenação*, o que implica na união de strings, ligando-as de ponta a ponta. Observe este exemplo:

```
>>> primeira = 10
>>> segunda = 15
>>> print(primeira + segunda)
25
>>> primeira = '100'
>>> segunda = '150'
>>> print(primeira + segunda)
100150
```

O operador * também funciona strings multiplicando o conteúdo de uma string por um inteiro. Por exemplo:

```
>>> primeira = 'Test '
>>> second = 3
>>> print(primeira * segunda)
Test Test Test
```

2.10 Requisitando valores ao usuário

Em alguns casos, nós podemos preferir que o valor para a variável seja inserido pelo usuário, por meio do seu teclado. Python possui uma função embutida chamada `input` que promove a entrada de dados via teclado[1]. Quando essa função é chamada, o programa pausa e espera o usuário digitar algo. Quando ele pressionar `Return` ou `Enter`, o programa volta à execução e o `input` retorna o que foi digitado como uma string.

```
>>> inp = input()
Qualquer besteira
>>> print(inp)
Qualquer besteira
```

Antes de pedir algo para o usuário, é uma boa ideia mostrar uma mensagem dizendo a ele o que digitar. Você pode passar uma string dentro do `input` para ser mostrada em tela antes da pausa para a entrada do dado: %não sei a melhor tradução para prompt

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

[1] No Python 2.0, essa função era chamada `raw\_input`

A sequência `\n` no fim da string representa *newline* (nova linha), que é um caractere especial que causa a quebra de linha. É por isso que a resposta do usuário aparece embaixo da pergunta.

Se você espera que o usuário digite um inteiro, você pode tentar converter o valor retornado para `int`, usando a função `int()`:

```
>>> prompt = 'Qual... a velocidade rasante de uma andorinha livre??\n'
>>> velocidade = input(prompt)
Qual... a velocidade rasante de uma andorinha livre??
17
>>> int(velocidade)
17
>>> int(velocidade) + 5
22
```

Porém, se o usuário digitar algo diferente de uma sequência de dígitos, ocorrerá erro nessa conversão:

```
>>> velocidade = input(prompt)
Qual...é a velocidade rasante de uma andorinha sem carga?
Seja mais específico, uma andorinha africana ou europeia?
>>> int(velocidade)
ValueError: invalid literal for int() with base 10:
            (conversão literal inválida para int() com base 10)
```

Veremos como lidar com esse tipo de erro depois.

2.11 Comentários

A medida que os programas se tornam maiores e mais complicados, eles ficam mais difíceis de ler. Linguagens formais são densas, muitas vezes é difícil olhar para um pedaço de código e descobrir o que está fazendo, ou porque.

Por esta razão, é uma boa ideia adicionar notas aos seus programas para explicar em linguagem natural o que o programa está fazendo. Essas notas são chamadas *comentários*, e no Python eles começam com o simbolo `#`:

```
# calcule a porcentagem da hora que se passou
porcentagem = (minutos * 100) / 60
```

Neste caso, o comentário aparece em uma linha própria. Você também pode colocar comentários no final de uma linha:

```
porcentagem = (minutos * 100) / 60     # porcentagem de uma hora
```

Tudo do # até o fim da linha é ignorado — não tem efeito no programa.

Comentários são mais úteis quando eles documentam características não-obvias do código. É razoável supor que o leitor possa descobrir *o que* o código faz; é muito mais útil que explique *porque*.

Esse comentário é redundante com o código e inútil:

~~~ {.python} v = 5 # atribuir 5 para v

Esse comentário contém informação útil que não está no código:

```
v = 5      # velocidade em metros/segundo.
```

Bons nomes de variáveis podem reduzir a necessidade de comentários, mas nomes longos podem dificultar a leitura de expressões complexas, portanto, há uma troca.

2.12 Escolhendo nomes de variáveis mnemônicos

Contanto que você siga as simples regras de nomeação de variáveis, e fuja de palavras reservadas, você tem muitas opções no momento de nomeá-las. Inicialmente essa escolha pode ser confusa tanto ao ler um programa quanto ao escrever o seu próprio. Por exemplo, os próximos três programas são idênticos em termos do que realizam, porém são muito diferentes quando você os lê e tenta entendê-los.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
horas = 35.0
taxa = 12.50
pagamento = horas * taxa
print(pagamento)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

O interpretador de Python vê todos os três programa *exatamente da mesma forma*, mas, os humanos veem e interpretam esses programas de formas bastante diferentes. Humanos vão entender mais rapidamente a *intenção* do segundo programa, pois o programador escolheu nome de variáveis que refletem sua função a respeito de qual dado será armazenado em cada variável.
~~~

Chamamos essas sábias escolhas de nomes "nomes de variáveis mnemônicos". A palavra *mnemônico*[2] significa "auxiliar de memória". Escolhemos nomes de variáveis mnemônicos para nos ajudar a lembrar o porquê criamos inicialmente essa variável.

Enquanto tudo isso parece ótimo, e é uma otima ideia usar nomes de variáveis mnemônicos, isso pode também atrapalhar a capacidade de um programador iniciante de analisar e entender o código. Isto acontece porque programadores iniciantes ainda não memorizaram as palavras reservadas (existem apenas 33 delas) e às vezes variáveis com nomes que são muito descritivos começam a parecer parte da linguagem e não apenas boas escolhas de nomes.

Dê uma breve observada na seguinte amostra de código que faz um loop em alguns dados. Nós iremos falar de loops em breve, porém por enquanto tente apenas entender o que isso significa:

```
for palavra in palvras:
    print(palavra)
```

O que está acontecendo aqui? Quais das palavras (for,palavra, in, etc.) são reservadas e quais são apenas nomes de variáveis? Python entende um nível básico de noção de palavras? Programadores iniciantes tem dificuldade em separar qual parte do código *tem* que ser iguais a esse exemplo e quais partes do código são simplesmente escolhas feitas pelo programador.

O código a seguir é equivalente ao anterior:

```
for fatia in pizza:
    print fatia
```

É mais fácil para o programador iniciante olhar esse código e saber quais partes são palavras reservadas definidas por Python e quais partes são simplesmente nomes de varáveis decididas pelo programador. É bem claro que Python não tem o básico entendimento de pizza e fatias e do fato de que uma pizza é um conjunto de uma ou mais fatias.

Mas se nosso programa é verdadeiramente sobre leitura de dados e procurar palavras nos dados, `pizza` e `fatia` são nomes de variáveis não-mnemônicos. Escolhê-los como nome de variáveis distraem do sentido do programa.

Depois de um curto período de tempo, você saberá as palavras reservadas mais comuns e começará a vê-las saltar aos seus olhos:

As partes do código que são definidas por Python (`for`, `in`, `print`, e `:`) estão em negrito e as variáveis escolhidas pelo programados (`palavra` e `palavras`) não estão
.
Muitos editores de texto estão atentos à sintaxe de Python e irão colorir as palavras reservadas diferentemente para lhe dar dicas para mantê-las separadas de suas variáveis. Depois de um tempo você começará a ler Python e rapidamente determinar o que é variável e o que é palavra reservada

[2] Veja https://en.wikipedia.org/wiki/Mnemonic para uma descrição maior da palavra "mnemônico".

2.13 Debugging

Nesse ponto, o erro de sintaxe que você está mais habituado a cometer é um nome inválido para variáveis, como `class` e `yield`, que são palavras-chave, ou `odd~job` e `US$`, que contém caracteres ilegais.

Se você põe um espaço em um nome de variável, Python pensa que são dois operandos sem um operador:

```
>>> nome ruim = 5
SyntaxError: invalid syntax

>>> mes = 09
  File "<stdin>", line 1
    mes = 09
            ^
SyntaxError: invalid token
```

Para erros de sintaxe, as mensagens que as indicam não ajudam muito. As mais comuns são `SyntaxError: invalid syntax` e `SyntaxError: invalid token`, nenhuma delas é muito informativa.

O erro de execução que você mais vai cometer é o "usar antes de definir"; que é tentar usar uma variável sem antes atribuir um valor para ela. Isso pode acontecer se você informar um nome de uma variável errado:

```
>>> principal = 327.68
>>> interesse = principio * rate
NameError: name 'principio' is not defined
```

Nomes de variáveis diferenciam maiúsculas de minúsculas. Assim, `LaTeX` não é o mesmo que `latex`.

Nesse ponto, a causa mais frequente de erro semântico será a ordem das operações. Por exemplo, para calcular $1/2\pi$, você pode se sentir tentado a escrever:

```
>>> 1.0 / 2.0 * pi
```

Mas a divisão vem primeiro, e você terá $\pi/2$, que não é a mesma coisa! Não tem nenhum jeito de Python saber o que você queria escrever, então nesse caso não haverá nenhuma mensagem de erro, você só vai ter a resposta errada.

2.14 Glossary

atribuição Uma declaração que atribui um valor a uma variável.
concatenação União de dois operandos de ponta a ponta.
comentário Informações em um programa destinado a outros programadores(ou qualquer pessoa que esteja lendo o código fonte) e não tem efeito sobre a execução do programa.

avaliar simplificar uma expressão executando as operações para gerar um único valor.
expressão uma combinação de variáveis, operadores e valores que representa um único valor de resultado.

ponto-flutuante Um tipo que representa a parte fracionária.
inteiro Um tipo que representa números inteiros..
palavra chave Uma palavra reservada que é usada pelo compilador para analisar um programa; você não pode utilizar keywords como `if`, `def`, e `while` para serem nomes de variáveis.

mnemônico Um recurso auxiliar de memória. Muitas vezes damos nomes variáveis de mnemônicos para nos ajudar a lembrar o que é armazenado na variável.
modulus operator Um operador, denotado com um sinal de porcentagem (`%`), que trabalha com inteiros e produz o restante quando um número é dividido por outro.
operando Um dos valores nos quais um operador opera.
operador Um símbolo especial que representa um cálculo simples como adição, multiplicação ou concatenação de string.

regras de precedência: O conjunto de regras que regem a ordem na qual expressões envolvendo múltiplos operadores e operandos são avaliadas.

declaração Uma seção de código que representa um comando ou uma ação. Até agora, as declarações que temos são atribuições e declarações impressas.
string Um tipo que representa sequencias de caracteres.
tipo Uma categoria de valores. Os tipos que vimos até agora são inteiros (tipo `int`), números com ponto flutuante (tipo `float`), e strings (tipo `str`).
valor Uma das unidades básicas de dados, como um número ou string, que um programa manipula.
variável m nome que se refere a um valor.

2.15 Exercícios

Exercício 2: Escreva um programa que use `inputs` para solicitar ao usuário seu nome e, em seguida, faça um cumprimento.

```
Digite seu nome: Chuck
Olá Chuck
```

Exercício 3: Escreva um programa que solicite ao usuário as horas e o valor da taxa por horas para calcular o valor a ser pago por horas de serviço.

```
Digite as horas: 35
Digite a taxa: 2.75
Valor a ser pago: 96.25
```

Não vamos nos preocupar em garantir que nosso pagamento tenha exatamente dois dígitos após a casa decimal por enquanto. Se você quiser, você pode utilizar com a função nativa do Python `round` função para arredondar corretamente o pagamento resultante para duas casas decimais.

Exercício 4: Suponha que executamos as seguintes declaração por atribuição:

```
Largura = 17
Altura = 12.0
```

Para cada uma das expressões a seguir, escreva o valor da expressão e o tipo (do valor da expressão).

1. `Largura//2`
2. `Largura/2.0`
3. `Altura/3`
4. `1 + 2 * 5`

Use o interpretador Python para verificar suas respostas.

Exercício 5: Escreva um programa que solicite ao usuário uma temperatura Celsius, converta para Fahrenheit, e mostre a temperatura convertida.