**Python**

**Curso 1 – Introdução a nova versão da linguagem:**

1. **Aula 1 – Introdução e instalação do python 3:**
   1. Para instalar o python na versão mais recente para o windows acesse esse site: <https://www.python.org/downloads/windows/>
   2. Nos Linux e macOS o python 2 e 3 geralmente já vem instalado.
      1. Caso não venha abra o terminal e use o comando *sudo apt-get install python3*.
      2. No mc entrar no site <https://brew.sh>, abrir o terminal e colar a linha de código que aparecerá. Essa linha irá instalar um gerenciador de pacotes (que no Linux já vem por padrão) e agora você poderá instalar o python3.
      3. Use o comando *brew update* para atualizar.
      4. Use o comando *brew install python3* para instalar o python3.
      5. Se quiser também pode entrar no site <https://www.python.org/> e baixar como no windows.
   3. Caso não possa instalar o python na sua máquina por qualquer motivo que seja, você pode utilizá-lo no navegador através do site: <https://replit.com>. Lá você também poderá escrever linhas de código.
      1. Esse site serve para diversas linguagens, não somente pytho.
      2. Python3 vem como primeira opção após clicar em “Start coding”.
      3. Este é um dos melhores sites para codar.
   4. Todas as funções em uma linguagem tem “()” depois de seu nome.
      1. Ex.: exit(), print(), help().
   5. A Função print() imprime o que estiver dentro de () na tela, ex.: print(“olá, mundo”). Essa frase será exibida quando o programa for executado.
   6. Utilizar o parâmetro*, sep=”caracter”* no print, faz com que todos os espaços que separam cada valor colocado no print seja substituído pelo que estiver dentro das “ ”.
   7. O parâmetro *end=”\n”* faz com que após o final da frase a linha seja quebrada e o que vier depois dele aparecerá na linha de baixo. Esse é o padrão
      1. Se deixarmos apenas *end=””* ele não quebra a linha e o que deveria ir para linha de baixo como o padrão, continua na mesma linha.
      2. Isso também funciona para o pointer de digitação, quando utilizamos esse end as >>> no terminal ficam logo no final da frase digitada.
   8. Podemos definir uma variável colocando o nome dela e o que ela receberá: *pais = “Italia”*.
      1. Sempre que a variável *pais* for chamada o que aparecerá será o nome *Italia*.
      2. Caso a gente *print(pais)* o resultado será exibido como *Itália* no terminal.
      3. Outro exemplo *print(pais, “ganhou”, 5, “títulos”, “mundiais”)*. Resultado: *Itália ganhou 5 títulos mundiais*.
   9. Podemos utilizar a função *type(variável)* para saber seu tipo. Ex.: *type(pais)*, o terminal exibirá: *<class ‘str’>*, ou seja, essa variável é uma string.
   10. Tipos de variáveis:
       1. String (str): texto. Sempre quando algo é escrito dentro de “ ” ela se torna uma string.
       2. Inteiro (int): Números inteiros. Quando uma variável recebe um número (sem vírgula), ela se torna do tipo inteiro.
       3. Float: Números flutuantes/reais. Quando uma variável recebe um real (com vírgula), ela se torna do tipo float.
       4. Boolean: Verdadeiro ou falso.
       5. Uma variável sempre tem um tipo.
       6. O python não precisa que você defina explicitamente qual é o tipo de uma string, ele já faz isso e à altera dinamicamente, diferentemente de outras linguagens como C++, por exemplo.
   11. Em python utilizamos o padrão Snake\_Case para nomear variáveis, ou seja, se o nome da variável for composto as palavras serão separadas por “\_”, ao invés de CamelCase, que o padrão de outras linguagens como JavaScript.
2. **Aula 2 – Lidando Com a Entrada do Usuário:**
   1. Para criar condições fazemos do mesmo modo que em outras linguagens, usando *if* e *else*.
      1. A diferença do python para as outras é que não precisa colocar {} para separar a condição em um bloco, o que conta é a organização/indentação do código.
      2. Depois de declarar qual a condição colocamos um “:” antes de começar a escrever o que acontece.
   2. Sempre que recebemos algo pelo input, essa informação vem em formato de *str*, caso queira que venha em um formato específico como *int* ou *float*, precisamos declarar isso com o *numero\_str =* *int(input(“Digite um número:”))*, ou colocando uma segunda variável e fazendo ela receber o input na conversão: *numero\_str = input(“Digite um número:”); numero = int(numero\_str)*.
   3. Quando utilizamos 1 “=” estamos atribuindo um valor a algo, quando utilizamos 2 “==” estamos fazendo uma comparação entre conteúdo de duas variáveis para ver se são iguais, e, quando usamos 3 “===”, estamos fazendo uma comparação de conteúdo e tipo de variável entre 2 variáveis. É um seletor estritamente rígido.
   4. O sinal de “+” quando colocado entre duas str irá concatenar ambas e imprimir uma palavra só com as duas palavras de cada variável, quando colocado entre números ocorrerá a soma e quando colocado entre número e str o código não funciona, a menos que convertamos o número de str para int/float.
   5. Syntax Sugar = nome dado à “técnica” de copiar várias vezes a mesma string como por exemplo *print(“20” \*10)* o que acontecerá é que será impresso 10 vezes o número “20”. Isso é muito útil para facilitar coisas que seria muito trabalhosas.
3. **Aula 3 – Testando Valores:**
   1. Podemos usar o *elif* *(condição):* quando temos mais de 2 opções de condições. Essa função fica depois do *if* e antes do *else*.
   2. Para melhorar a legibilidade do código podemos criar variáveis que recebam as condições impostas nos *if’*s e nos *elif’*s, e colocar essas variáveis no lugar das condições, facilitando sua leitura.
   3. Operadores de comparação:
      1. < - menor que;
      2. > - maior que;
      3. <= - menor ou igual a;
      4. >= - maior ou igual a;
      5. == - igual a;
      6. != - diferente de.
   4. No python é aceita as formas com e sem “()” nas condições, no entanto, utilizamos () para deixar mais claro qual é a condição, principalmente ao utilizar os operadores lógicos *AND* e *OR*.
4. **Aula 4 – A sequência do Jogo:**
   1. Para que um pedaço do código se repita diversas vezes podemos, ao invés de copiar e colar esse código diversas vezes, utilizar a função *While*, que faz com que o código se repita uma determinada quantidade de vezes.
   2. Podemos escrever uma única frase no print e depois formatar pontos específicos utilizando: *print(“frase {} completa {}”, format(item\_1, item\_2))*. Dessa forma os {} serão substituídos pelo valor das variáveis no *format*, o primeiro trocado pela primeira variável a assim subsequentemente.
      1. Isso é bom pois facilita na hora da escrita, uma vez que podemos escrever a frase inteira de uma vez e no final apenas pedir para que os {} sejam substituídos pelos valores respectivos da ordem.
      2. Isso evita com que tenhamos que ficar abrindo e fechando aspas, colocando vírgulas dentre outros.
5. **Aula 5 – Iterando de Maneira Diferente:**
   1. Existe outro sistema de laços chamado *for* que funciona para repetir algo uma determinada quantidade de vezes, mas sem definir condições.
      1. Para utilizar essa função precisamos logo após o *for* determinar uma variável contadora (geralmente utilizado o *i*) para armazenar a quantidade de repetições.
      2. Após a variável precisamos definir que o alcance/área de repetição desse código, ou seja, se ele vai repetir de qual número a qual número e qual é o step dele, portanto, de quantas em quantas casas ele irá pular (padrão = 1).
      3. Como o “alcance” é uma função ele será seguido de ().
      4. Para exemplificar fica assim: *for i in range (1, 10, 2): código*.
      5. Diferentemente do *While*, o for já define a variável quando colocada em sua sequência, enquanto no *While* temos a necessidade de defini-la antes.
      6. Como pudemos ver no exemplo, a única parte que possui () é na função range, também sendo essa uma diferença sintática entre ambas as funções.
      7. Podemos juntar o for com um *input* para definir a quantidade de vezes que algo será repetido, deixando com que o usuário possa definir da maneira que quiser.
      8. Quando colocamos um número máximo para repetir, precisamos lembrar sempre que será repetido até o número anterior ao colocado. No exemplo acima pedimos para ir até o 10, porém, como é essa lógica que se segue, o código se repetirá até o 9.
      9. Para corrigir esse problema podemos colocar o 11 no final ou adicionar um *+1*, para que se tenha o mesmo resultado.
   2. Caso a sua condições do While (em um jogo de adivinhação, por exemplo) seja satisfeita e você tenha acertado o número, o jogo não irá parar até que o número de tentativas escolhido tenha se completado.
      1. Porém isso não faz sentido, uma vez que o jogo tenha sido ganho, não tem porquê continuar. Portanto, para resolver esse problema, podemos utilizar a função *break* no final do *if* de acerto, desse modo o jogo para de rodar quando a condição for satisfeita.
      2. Existe também o *continue* que é o oposto do break. Ele vai terminar a interação (de um aviso, por exemplo), mas não vai quebrar o laço e terminar o programa.
      3. O programa continua seguindo a partir do aviso inicial, porém pulando aquela interação, ou seja, parte para o próximo loop.
   3. Sobre interpolação de str’s:
      1. Utilizando o *format(1,2)* depois do *print* fazemos com que as {} sejam substituídas pelos valores respectivos, como já vimos. Contudo, nós podemos determinar qual parâmetro do format será exibido em qual {} colocando números dentro deles.
      2. O primeiro é sempre representado pelo ***0*** e os próximos a partir disso.
      3. Exemplo: *print(“O {} tem um total de {} mundiais, já o {} tem {} mundiais”, format(palmeiras, 0, corinthias, 3)*. Ele irá imprimir os valores respectivos por padrão, mas e se quisermos mudar?
      4. *print(“O {2} tem um total de {3} mundiais, já o {0} tem {1} mundiais”, format(palmeiras, 0, corinthias, 3)*. Determinando as interpolações assim, os valores são trocados, apresentando primeiro o corinthians e depois o palmeiras, diferentemente do anterior.
   4. Podemos formatar números e quantidades de espaços utilizados por palavras frases formatadas colocando dentro dos {} os:
      1. **:** = indica formatação;
      2. **.número** = indica quantidade de casas depois da vírgula/ponto;
      3. **F** = Indica que é um número float;
      4. **D** = Indica número inteiro;
      5. **Número depois dos :** = Indica quantas casas o conteúdo do format ocupará;
      6. **Número antes do número depois dos :** = Indica qual é o número que irá ocupar todas as casas que em branco.
   5. Comparações do python 2 com o 3: <https://pyformat.info/>
   6. Podemos também interpolar diretamente dentro dos {}, sem utilizar o .format. O nome dessa funcionalidade é *f-strings* ou *formatted string literals*.
      1. Para usar isso basta colocar toda a string entre aspas simples ‘ ‘ e um f antes delas, ficando assim: *print(f’texto {qualquer} de {exemplo}’)*. Assim as informações recebidas pelas variáveis *qualquer* e *exemplo*, serão substituídas no texto printado.
      2. Isso também nos permite adicionar funções de formatação dentro, como *{qualquer.lower()}* para colocar todas as letras do valor da variável em minúsculas ou *.upper()* para em capslock e assim por diante.
6. **Aula 6 – Gerando Números Aleatórios:**
   1. Podemos adicionar bibliotecas nos nossos programas para facilitar nossa vida e agilizar processos.
      1. A que vamos utilizar no nosso projeto de jogo de adivinhação é a *Random*, sendo ela a biblioteca que nos proporciona diversos números ou o que quer que seja aleatoriamente.
      2. Para importar a biblioteca usamos *import nome\_da\_biblioteca*.
   2. Além de importarmos bibliotecas inteiras nós podemos importar também funcionalidades específicas de cada biblioteca, para que desse modo nosso programa não fique sobrecarregado ao fazer download de muitas informações de várias bibliotecas diferentes.
      1. No nosso caso vamos importar apenas a função Randint da biblioteca Random para gerarmos um número inteiro aleatório.
      2. Para importar apenas uma função usamos *from nome\_da\_biblioteca import função\_desejada*.
   3. O próprio python já vem com várias bibliotecas inclusas, sendo elas as suas funções, como o próprio *print()*, por exemplo.
      1. Para saber mais sobre quais são suas funções nativas basta procurar python builtins no google e acessar o doc do site oficial da versão python que deseja ver.
   4. A biblioteca Random é na verdade um pseudo-random pq não gera um número verdadeiramente aleatório, o número é gerado a partir de uma *seed* que podemos definir caso a gente queira, por exemplo, caso setamos a seed 100, sempre que rodarmos o código o número Random aleatório que será gerado será o 19 indefinidamente.
7. **Aula 7 – Nível e Pontuação:**
   1. Podemos utilizar condições if para fazer menus de seleções por números.
   2. Para evitar que a pontuação fique negativa caso o cálculo dê um número negativo, utilizamos a função *abs()* de absoluto, o que faz com que qualquer resultado que dê, fique positivo.
   3. Utilizando a função *round()* nós podemos arredondar os números floats.
   4. No python 3 os valores x.5 são arredondados para o número par mais próximo. Ex.: 3.5 – 4.5 = 4 – 4.
   5. Para fazer uma divisão que nos retorne número float podemos usar “/” para fazer uma que retorne número inteiro usamos “//”. Ex.: 3 / 2 = 1.5; 3 // 2 = 1.
8. **Aula 8 – Organizando Nosso código:**
   1. Podemos importar arquivos que nós criamos em python para outros arquivos e executa-los como funções.
   2. Para criar uma função precisamos colocar *def* *nome\_da\_função():*
   3. Podemos criar funções para códigos inteiros, no nosso caso dos jogos, e executar eles quando escolhermos o jogo no lobby.
   4. O problema de transformar arquivos em funções é que não podemos mais executar aquele arquivo diretamente, apenas se for pelo arquivo que está importando ele.
      1. Para corrigir isso precisamos colocar um *if (\_\_name\_\_ == “\_\_main\_\_”): nome\_da\_função()*.
      2. Fazendo isso o arquivo poderá ser executado individualmente.
9. **Aula 9 – Comparando Python com C:**
   1. Python não precisa ser compilado antes de ser executado/interpretado como na linguagem C, por exemplo.
   2. No C precisamos criar o executável compilando o arquivo principal do programa.
   3. Python pode ser portado para qualquer OS e ele irá rodar igualmente em todos, C não.

**Curso 2 – Avançando na Linguagem:**

1. **Aula 1 – Preparando o Jogo da Forca:**
   1. Podemos utilizar a palavra/comando *not* para indicar que algo não está acontecendo.
      1. Um exemplo é no While no jogo que estamos fazendo agora: *While(not enforcou and not acertou):...*. Ou seja, enquanto não enforcou o boneco e não acertou a palavra da forca, repete o loop.
      2. Operador lógico *and* faz com que, no caso do While, o laço se repita enquanto não está enforcado **E** não acertou, ou seja, se apenas uma condição for cumprida o loop se encerra.
   2. Bool recebem valores como 0, “ ” e none e consideram como falsos, mas o contrário também é verdade, ou seja, 1, “teste” e true são considerados como verdadeiros.
2. **Aula 2 – Manipulando Strings:**
   1. A função *find(“”)* serve para procurar uma letra ou palavra específica em uma string.
   2. Podemos usar o for para criar loops e encontrar as letras que pedimos em mais de uma posição na palavra e pedir para que ele diga qual a letra e em qual posição está.
   3. Usar a função .*lower()* depois de uma string/input fará com que essa string toda seja convertida para lowercase, mesmo que tenha sido escrita em capslock.
   4. *Capitalize()* faz a primeira letra de cada palavra da str ser maiúscula.
   5. *Upper()* faz todas as letras ficarem em caps.
   6. *Endswith(“str”)* retorna true ou false se o final da str terminar com o valor colocado nos ().
   7. *Startswith(“str”)* igual ao anterior, mas no início.
   8. *Strip()* retira os espaços anteriores e posteriores da string caso haja.
   9. Importante lembrar que todas essas funções não alteram a str original, mas sim geram uma nova str com a modificação colocada.
3. **Aula 3 – Conhecendo e Trabalhando com Listas:**
   1. Para fazer com que o python se lembre de uma estrutura que já foi utilizada, como no jogo da forca, podemos usar listas.
      1. Para saber mais a respeito de listas, segue a documentação: <https://docs.python.org/3/tutorial/datastructures.html>
   2. Utilizando listas ou strings podemos perguntar para o python se determinado valor ou letra existe naquela str ou palavra.
      1. Como exemplo podemos fazer:
      2. *Valores = [1, 2, 3, 4, “x”]*
      3. *3 in valores*
      4. Python responde: true
      5. *6 in valores*
      6. Python responde: false
      7. *“a” in “banana”*
      8. Python responde: true
      9. *“l” in “banana”*
      10. Python responde: false
   3. Além do visto acima, também podemos perguntar qual letra/valor está em determinada posição.
      1. Seguindo a variável de *valores* acima:
      2. *Valores[0]*
      3. Python responde: 1
      4. *Valores[4]*
      5. Python responde: ‘x’
   4. *Min(valores)*: Exibe qual é o menor valor de uma lista.
      1. **OBS.:** O comando precisa que todos os valores da lista sejam do mesmo tipo, nesse caso, a nossa variável *valores* não poderia ter o *“x”* para que funcionasse, apenas números.
   5. *Max(valores)*: Exibe qual é o maior valor de uma lista.
      1. **OBS.:** O comando precisa que todos os valores da lista sejam do mesmo tipo, nesse caso, a nossa variável *valores* não poderia ter o *“x”* para que funcionasse, apenas números.
   6. *Len(valores)*: Exibe o tamanho da lista, ou seja, número de caracteres presentes em uma lista.
   7. *Clean(valores)/valores.clean()*: Limpa todos os caracteres de uma listagem.
   8. *Valores.append(n)*: Adiciona o número desejado na última posição da listagem.
   9. *Pop(valores)/valores.pop()*: Devolve (printando) e remove o último elemento da listagem.
   10. *Reverse(valores)/valores.reverse()*: inverte a ordem dos valores da listagem.
   11. Lembrando que todas essas funções e muito mais se encontram no site listado acima.
   12. *Valores.indice(o\_que\_buscas)*: Mostra a posição do elemento buscado. Lembrando que a contagem começa no 0.
       1. Se o que buscas não está na lista, ele retorna um erro. Por isso é importante ter uma condição para primeiro verificar se existe o que está buscando ou não. Caso não, retorne uma mensagem dizendo que não.
4. **Aula 4 – Conhecendo e Trabalhando com Tuplas:**
   1. Tuplas também são listas e bem parecidas com elas, porém, elas não podem ser alteradas.
   2. Para definir listas utilizamos [] depois da variável, no caso de tuplas utilizaremos ().
   3. Basicamente não podemos alterar tuplas com .pop() dentre outras funções.
   4. Podemos criar várias variáveis que recebem tuples e colocar todas elas dentro de uma lista:
      1. *P1 = (3, 6)*
      2. *P2 = (“Nico”, 39)*
      3. *P3 = (37, “Flávio”)*
      4. *Line = [p1, p2, p3]*
      5. *Print(line)*
      6. Python exibe: [(3, 6), (‘Nico’, 39), (37, ‘Flávio’)]
   5. Podemos pedir para ele exibir só um dos tuples, como uma lista normal:
      1. *Line[0]*
      2. Python exibe: (3,6)
      3. *Line[1]*
      4. Python exibe: (‘Nico’, 39)
   6. Além disso, também podemos pedir para que ele mostre apenas um dos valores de um tuple:
      1. *Line[1][0]*
      2. Python exibe: ‘Nico’
      3. *Line[1][1]*
      4. Python exibe: 39
   7. Podemos converter listas para tuple e tuple para listas:
      1. *Lista\_frutas = []*
      2. *Lista\_frutas.append(“abacaxi”)*
      3. *Lista\_frutas.append(“banana”)*
      4. *Print(Lista\_frutas)*
      5. Python exibe: [‘abacaxi’, ‘banana’] **- lista**
   8. Quando não quiser mais que essa lista seja modificada:
      1. *Nova\_lista\_tuple = Tuple(Lista\_frutas)*
      2. *Print(Nova\_lista\_tuble)*
      3. Python exibe: (‘abacaxi’, ‘banana’) **- tuple**
   9. O contrário também é verdade:
      1. *Nova\_lista\_frutas = list(nova\_lista\_tuple)*
      2. *Print(Nova\_lista\_frutas)*
      3. Python exibe: [‘abacaxi’, ‘banana’] **– lista**
   10. Ambas coleções permitem que você adicione mais de um elemento igual a algum que já seja existem nelas, o que é um problema quando se está trabalhando com CPF’s por exemplo.
   11. Para solucionar esse problema, existe uma outra coleção chamada ***set***.
       1. Para determinar que a coleção é um *set*, utilizamos {}, ao invés de [] e () como no caso das listas e tuples, respectivamente.
       2. Diferentemente das outras coleções, precisamos chamar a função *add()* ao invés de *append()* para adicionar um item à coleção.
       3. Com essa coleção e função, sempre que formos adicionar um elemento que já existe na coleção, dará um erro.
       4. A coleção do *set* não é uma sequência como as outras 2 coleções, portanto, ela não possui um índice, deste modo não conseguimos pedir para que apenas um elemento em determinada posição seja exibido, como nos casos anteriores *coleção[0]*: Não exibe nada.
   12. Um set é uma coleção não ordenada de elementos. Cada elemento é único, isso significa que não existem elementos duplicados dentro do set.
   13. Todas essas coleções são boas, mas e se você tiver uma escola onde todos os instrutores estão junto com suas respectivas idades e você precisa saber a idade de um deles específico, mas não sabe em qual posição da coleção ele se encontra? Para resolver esse problema utilizamos a coleção ***dictionary***.
       1. Os *dictionaries* utilizam uma estrutura similar a do *set*, sendo inicializada pelos {}.
       2. A diferença dentre eles é o modo em que os valores são colocados.
       3. No *dictionary* os valores sempre são em pares e divididos por ***:***. Antes do ponto é a chave e depois do ponto é o valor.
       4. Desse modo ficando assim:
          1. *Instrutores = {‘Nico’ : 39, ‘Flávio’ : 37, ‘Marcos’ : 30}*
       5. Agora, caso você queira saber a idade de um instrutor, mas não sabe em qual posição da coleção ele está para encontra-lo assim: *instrutores[n]*, você pode simplesmente digitar o nome do instrutor desejado e a idade dele será impressa, assim: *instrutores[‘Flávio’]*.
5. **Aula 5 – Implementando o Encerramento do Jogo:**
   1. Podemos colocar valores booleanos com condições dentro de variáveis. Ex.:
      1. *Letras\_acertadas = [“\_”, “\_”, “\_”, “\_”, “\_”, “\_”]*
      2. *Acertou = “\_” not in letras\_acertadas*
      3. Nesse caso acima, quando não houver mais *“\_”* dentro da variável *letras\_acertadas*, a variável acertou receberá o valor *true*, mas, enquanto isso não acontecer e ainda tiver *“\_”* dentro de *letras\_acertadas*, a variável *acertou* terá valor *false*.
      4. Esse tipo de variável é bastante usada com laços *While* para verificação.
   2. Se tivermos uma variável contadora e quiser adicionar +1 nela, existem dois métodos para fazer isso:
      1. *Contador = 0*
      2. *Contador = contador + 1*
      3. *Contador += 1*
   3. Para as condições cujo valores de respostas para acontecer são booleanos, não precisamos colocar *if(acertou == true):*, podemos simplesmente colocar *if(acertou)* que já significará a mesma coisa que antes.
   4. No jogo da forca, para colocar a quantidade *“\_”* referente a quantidade de letras da palavra secreta, podemos fazer um laço *for*:
      1. *Letras\_acertadas = []*
      2. *For letra in palavra\_secreta:*
      3. *Letras\_acertadas.append(“\_”)*
   5. Outro método de fazer a mesma coisa que acima é basicamente escrever todo o laço dentro de uma única linha na própria lista:
      1. *Letras\_acertadas = [“\_” for letra in palavra\_secreta]*.
      2. O nome desse tipo de técnica é *list comprehension*.
   6. Também podemos utilizar *list comprehension* com condições.
      1. Caso você tenha uma lista de números inteiros e queira pegar somente os pares, precisa fazer uma condição:
         1. *Inteiros = [1,3,4,5,7,8,9]*
         2. *Pares = []*
         3. *For numero in inteiros:*
         4. *If numero%2 == 0:*
         5. *Pares.append(numero)*
      2. Basicamente significando que para cada número em inteiros, se ele dividido por 2 tiver resto 0, adicionar a lista de pares.
      3. Porém, podemos utilizar uma list comprehension para dizer a mesma coisa:
         1. *Inteiros = [1,3,4,5,7,8,9]*
         2. *Pares = [x for x in inteiros if x % 2 == 0]*
      4. O que deixa o código todo muito mais enxuto e melhor de visualizar.