復習

プログラミング1

(第1回) 卓上プログラミングによる開発設計概観、Pythonインタプリタの起動と逐次処理・変数の利用

1. プログラミングとは何か？
    1. プログラムの特徴
    2. （プログラミングにおける2大原則）
    3. （プログラミングを円滑に進めるための周辺技術）
2. 演習1：教科書のコードを動かしてみる
3. 演習2：オブジェクトと式と型
4. 演習3：変数と等号の利用、実行の様子
5. シラバス
6. 授業方針
7. 宿題・補足

実現したいことを理解し、手順として整理し、プログラミング言語で記述（翻訳）すること。ペアプロで互いに教え合おう。

ターミナル＋Pythonインタプリタを使った作業工程に慣れよう。Python2と3の違いに注意。

代表的な型（int, float, str）、型の確認方法と演算子を覚えよう。

プログラミングにおける等号は、(1)右辺を評価(実行)して、(2)結果を変数に保存する。

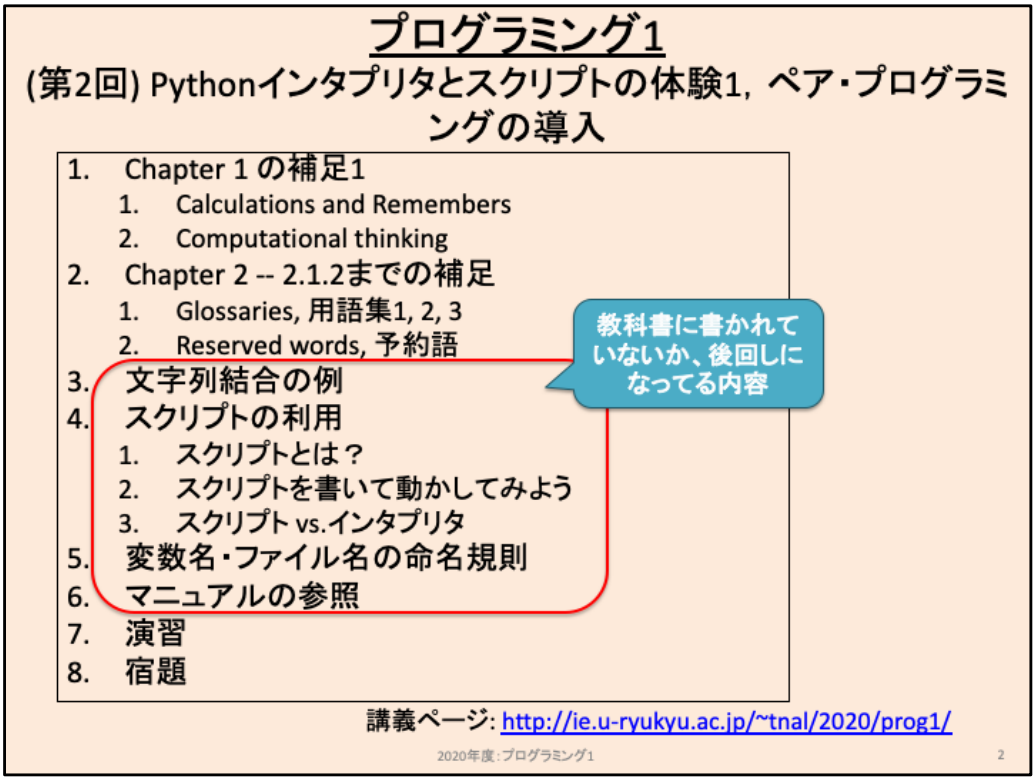
達成目標、評価方法等について必要な時に参照。

疑問に感じた点はどんどん質問しよう。

教科書・参考サイト参照しつつ、手を動かそう。

講義ページ: http://ie.u-ryukyu.

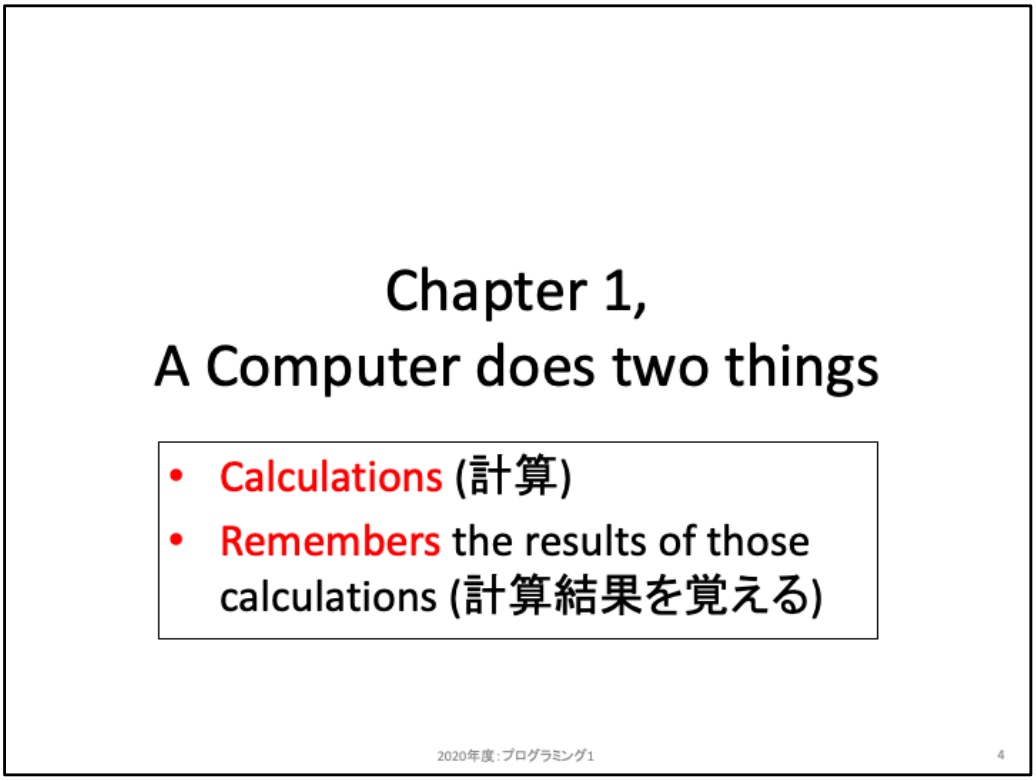
1



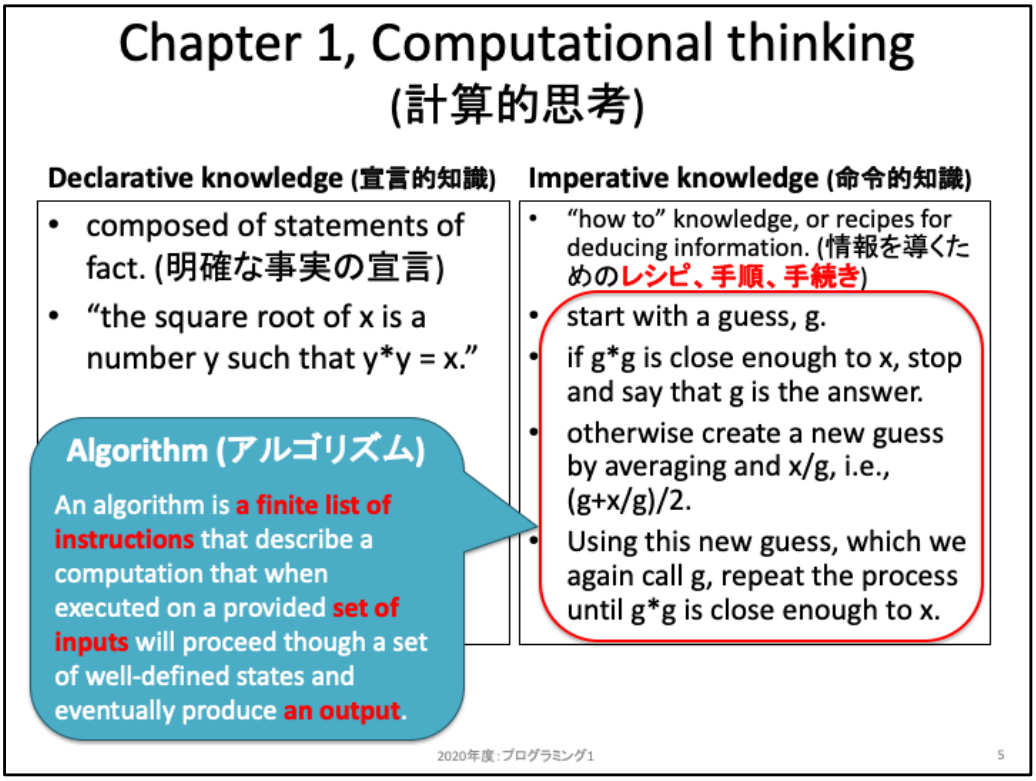
赤枠は教科書には書かれていないか、もしくは後回しになっていますが重要なので早めに今回扱います。

Chapter 1 の補足1

複数回に分けて補足します



コンピュータは「計算すること」「その計算結果を覚えること」の2つのことしか実行できない。



左の宣言的知識で述べている平方根の定義は、定義として正しい。しかしこの定義からは平方根を求めることはできない。

右の命令的知識で述べているレシピ・手順・手続きは、平方根を求めるための手続きとして書き下されており、近似的な平方根を求めることが可能。このように「十分に定義された入力が与えられ、有限個の命令群で用意された手続きにより出力を求めることができる手続き」のことをアルゴリズムと呼ぶ。より詳細は2年次の必修講義「アルゴリズムとデータ」にて学ぶ。ここでは「入力と出力を明確にし、それをどのように処理するのかという手続きを考える必要があること」ぐらいを心に留めておこう。

プログラミング1

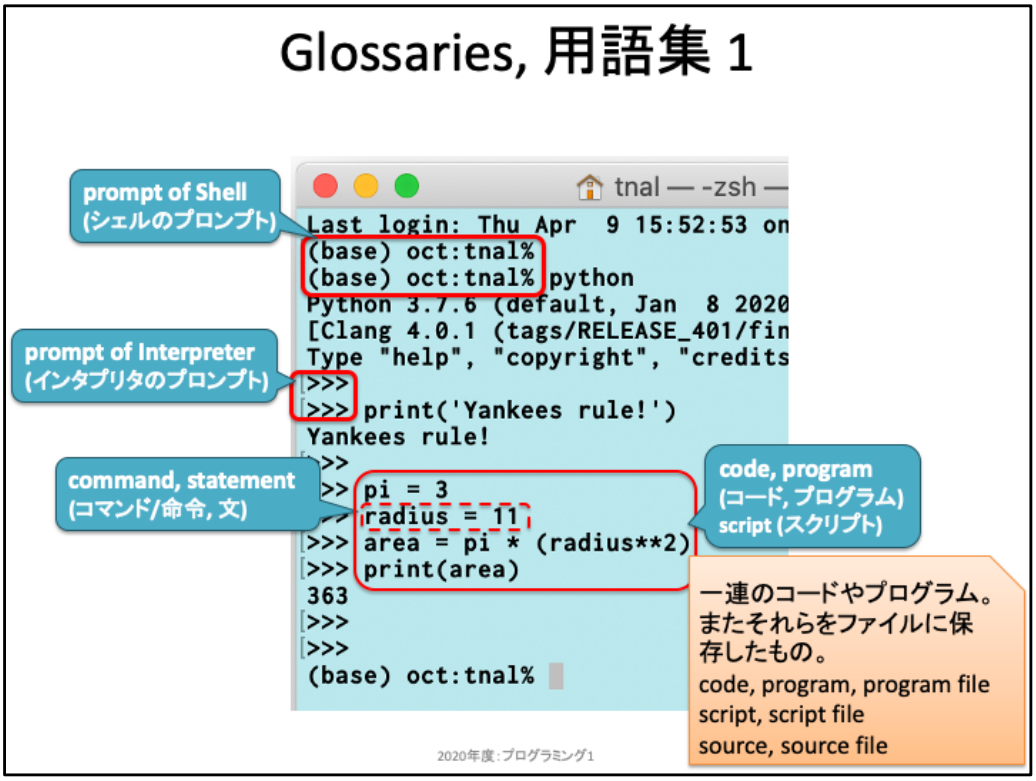
(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

欲しい出力を得るためのレシピを考える必要がある。レシピ≒アルゴリズム。

Chapter 2 -- 2.1.2までの補足

Glossaries, 用語集
Reserved words, 予約語

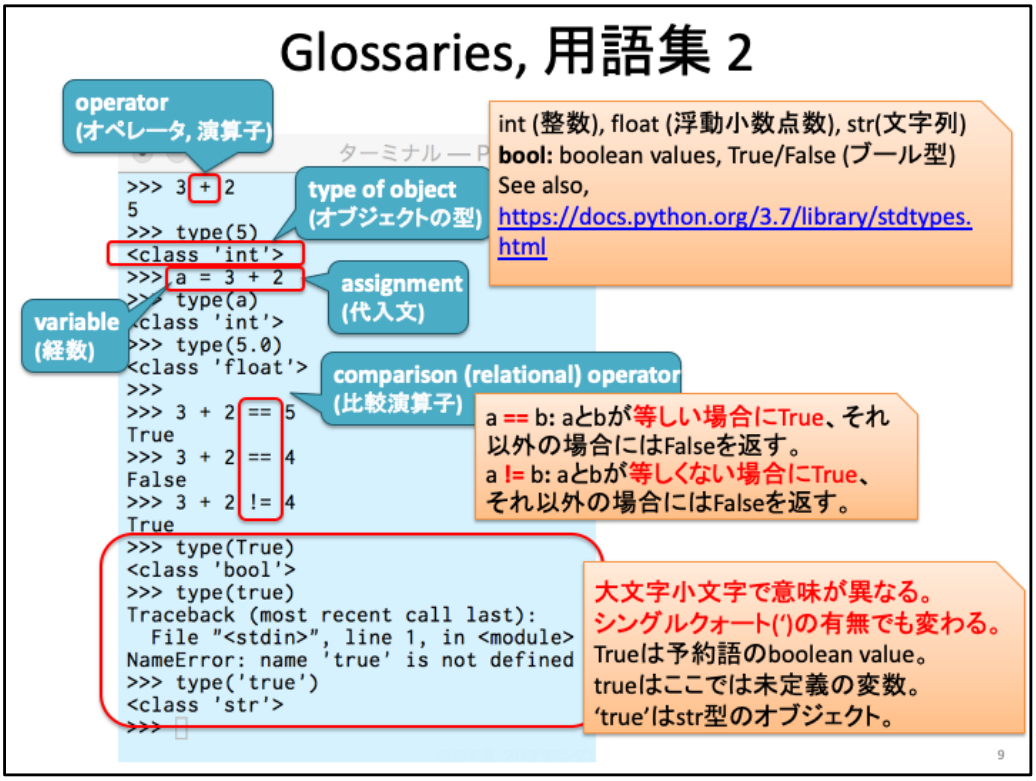


ターミナル上でアプリケーション（シェルやPython等）とやり取りを行う際、アプリ側が入力を受け付けられる状態か否かを明示するために用いられるのが「プロンプト」である。ターミナルを起動した直後の状態からプロンプトが出力される。具体的な命令を入力せずにEnterキーだけを入力すると、キーが入力される度にプロンプトが出力される。この「プロンプトが返ってくる」ことを通して、こちらの入力を受け付けているということが分かる。

ターミナルを起動した直後に出力されるプロンプトを、上記では「シェルのプロンプト」と書いている。これは、ターミナルを起動すると自動的にシェルが起動され、実際に応答しているのはシェルだからだ。シェル上でpythonと命令を入力すると「Pythonインタプリタ」が起動し、プロンプトの出力形式が変わる。「>>> 」の状態ならPythonインタプリタが入力を受け付けている状態である。このようにプロンプトから「今何とやり取りしているのか」を意識しながら取り組もう。

なお後述するが、シェルやPythonインタプリタに対する入力は「標準入力（standard input, stdinと省略）」、逆に得られる出力を「標準出力（standard output, stdout）」と呼ぶ。「標準」と付けている理由は、入出力をする経路が複数あるからである。例えば、ターミナルウィンドウを複数立ち上げているとしよう。それぞれのウィンドウでPythonインタプリタを起動している場合、それぞれのインタプリタはどのように入力を受け、またどのように出力したら良いだろうか。このような混乱をしなくて済むように、特に何も指定しなかった場合にはインタプリタを立ち上げたウィンドウを対象とするように紐付けられている。標準で用意されている入出力先のことを標準入力、標準出力と呼ぶ。

Pythonインタプリタが処理を実行する際の単位をコマンドや命令と呼び、それらを集合をコードやプログラム、またソース等と呼ぶ。また、コードやプログラムをファイルとして保存したものをスクリプト（スクリプトファイル）、ソース（ソースファイル）と呼ぶ。文脈に応じて意味が異なることもあるが、他人とのやり取り中における「ソース見せて」という文は、「書いたコードを見せて」「保存したコードを見せて」と同等の意味で使われる。



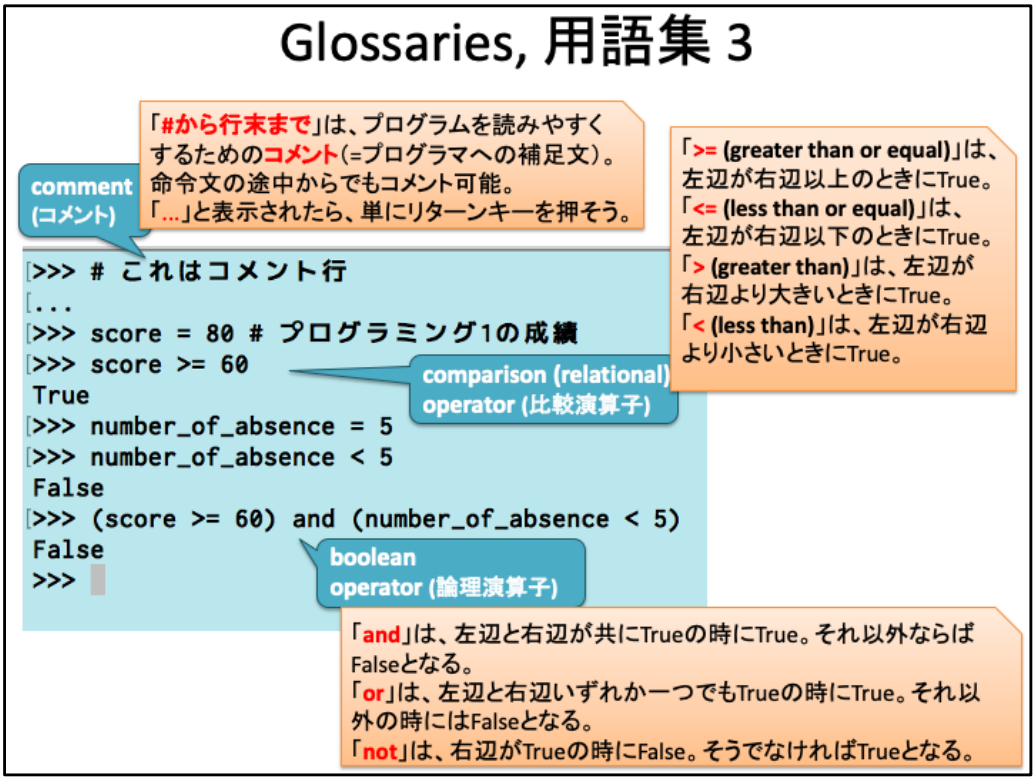
基本的な型、演算、type()関数で型を調べられることを覚えておこう。

==や!=は比較演算子と呼ばれており、演算子の左右が等しいか否かを確認するために用いられる。例えば「先着100人までプレゼントする」ようなコードを書こうとすると、応募のあったユーザを順番に並べて一人ずつカウントする。そのカウントした値を変数countに保存しておき、「count == 100」の比較結果が真であれば、100人分処理し終えたということを判断できる。比較演算子の結果は真偽のみであり、True もしくは False というbool型（boolearn, ブーリアンとも呼ばれる）のリテラルで返される。これは文字列ではない点に注意。例えば、

>>> True == 'True'

において、左辺はboolean型のTrue（真）というリテラルであり、右辺はstr型のリテラルである。異なるリテラルであるため、比較結果は False となる。

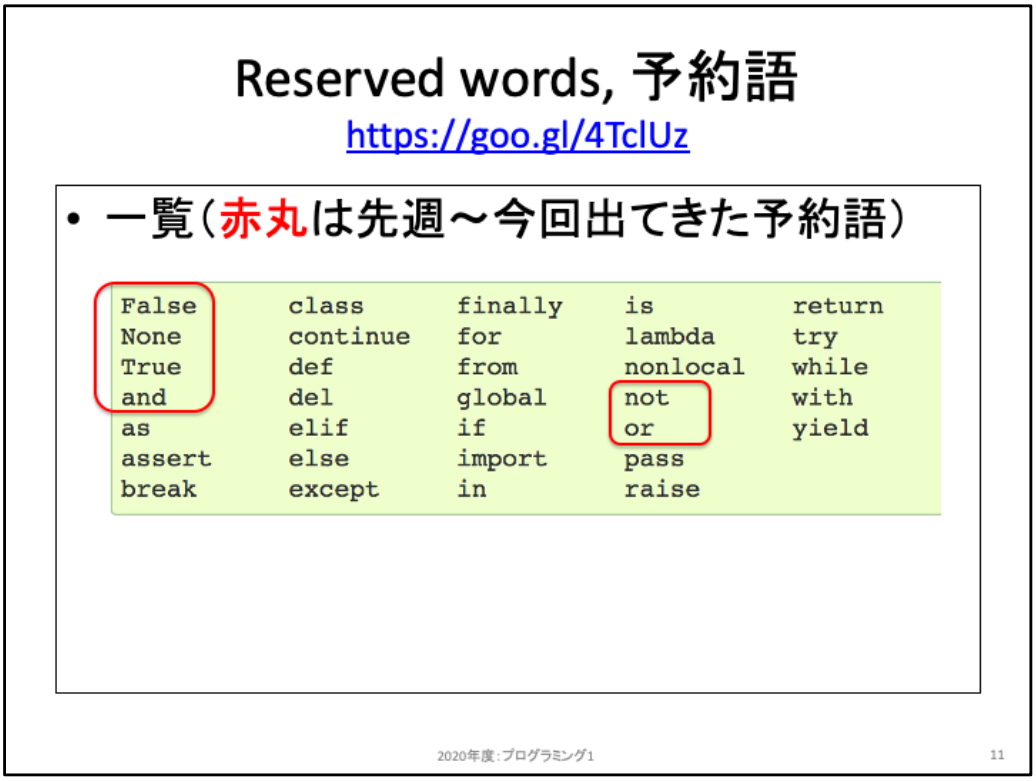
print()やtyp()のように、「名称(引数)」の形式で実行する命令を「関数（functions）」と呼ぶ。数学における関数の場合、例えば y = 2x + 1 という1次関数では、変数xの値を具体的に決めると出力yを算出することができる。この変数xに相当する部分、すなわち入力を指定する部分をプログラムでは「引数(arguments）」と呼ぶ。プログラムにおける関数は、引数で与えられる情報を元に、何かしらの処理を行う。具体的には、print()関数では「引数として与えられたリテラル、もしくは変数の中身を標準出力に書き出す」し、type()関数では「引数として与えられたリテラル、もしくは変数の中身の型を調べ、返す」という処理を実行する。関数は自身でも作ることが可能であり、後日取り扱う。興味のある人は教科書で予習してみよう。



命令文に # が含まれていると、そこから行末までは「コメント」として扱われる。コメントは人間がメモをしておくために用いる。また、一時的にコードを実行したくないが削除したくない場合に、コードの冒頭に#をつけることで該当行をコメントとして解釈させることもあり、このような行為は「コメントアウト（comment out）」と呼ぶ。

>, >=, <, <= は数値を用いた比較演算子であり、「より大きい」「以上」のようにその値を含むのかどうかを区別できるようになろう。この使い分けを間違うと「100人に対しプレゼントする」つもりでも99人までしか処理できなかった、ということになる。よくあるミスであり、このようなミスを発見しやすくするための手法を境界テストと呼ぶ（後日やります）。

and, or, not は論理演算子と呼ばれており、左右のリテラルがbool型であることを前提として処理する。例えば「条件1も条件2も揃っているのでステージクリアと見做す」ような場合には and 演算子による確認が向いている。or, not についてもどういう用途がありそうか、考えてみよう。



ここまで既に多数の用語について取り上げた。これら全てを一度に理解し覚えておくことは求めていない。少しずつ理解し、忘れた場合には教科書等を振り返り、理解できる部分を増やしていこう。

プログラミング1

(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

レシピを記述するための道具（基本的な型・算術演算子・比較演算子・論理演算子）を使えるようになろう。

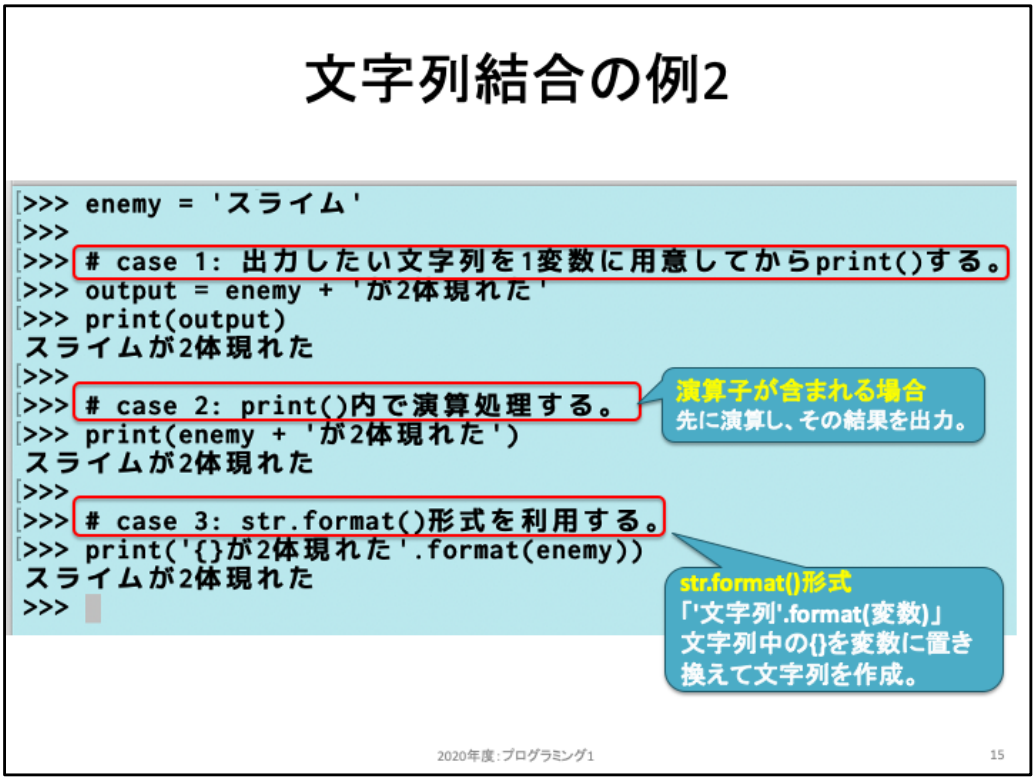
文字列結合の例
（教科書に書かれてません）

文字列結合の例1

```
>>> # 文字列の演算
>>> 'スライム' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> 'スライム' + 'が2体現れた'
'スライムが2体現れた'
>>> enemy = 'スライム'
>>> enemy + 'が2体現れた'
'スライムが2体現れた'
>>> output = enemy + 'が2体現れた'
>>> print(output)
スライムが2体現れた
>>>
```

数値と文字列を直接足すことはできない
TypeError: must be str, not int
型エラー: str型であるべき

文字列同士の足し算は、結合になる。



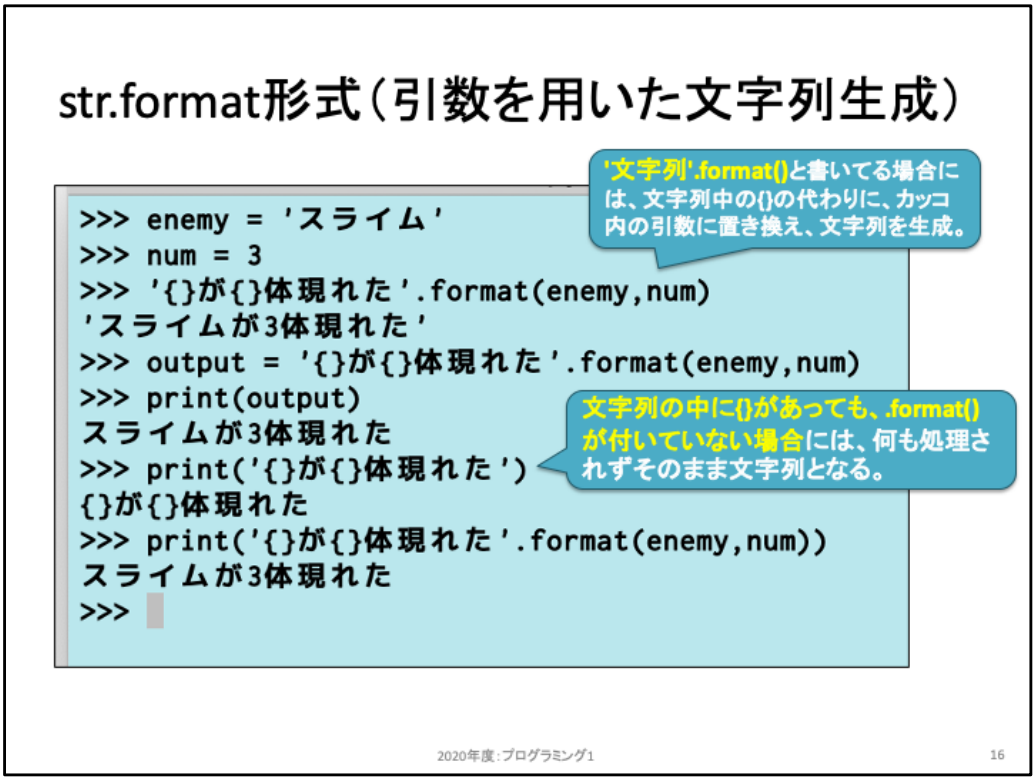
ここでは文字列結合の例を3つ示している。
1つ目は、前のスライド最後のコードを再掲している。文字列を保存した変数に対して文字列リテラルを結合している例である。

2つ目は、引数内に演算子が含まれるケースを示している。このように演算子が含まれる場合、演算子がなくなるまで演算を実行し、その結果を引数として利用する。

3つ目は、とても見づらいがPython特有の文字列作成方法である。いくつか新しい要素が出てきているため、分解して考えてみよう。
(1) '{}が2体現れた' は、str型の文字列を表している。
試しに '{}が2体現れた' のみを引数として指定した場合には、そのまま出力されるはずである。

(2) (1)の後ろに続く .format(引数) は、直前の文字列における {} を引数に置き換えた文字列を生成する。
例えば、
>>> '{}'.format(1)
は、{}を数字の1に置き換えた文字列を生成する。
>>> '{}'.format(enemy)
は、変数enemyが何か値を保存しているならば、その中身を{}の部分に置き換え、文字列を生成する。
>>> '{}が{}体現れた'.format(enemy,2)
は、1つ目の{}を変数enemyの中身に置き換え、2つ目の{}を数字の2に置き換えた文字列を生成する。

このような一見複雑に見える文字列結合の手段を用意しているのは、単に +演算子による結合を用いた場合にわかりづらくなるケースがあるからだ。例えば、
>>> '1+1' + 'は' + '2' + 'です'
というコードを人間が見たとき、どの部分がリテラルなのかは直感的には分かりづらい。これに対し、
>>> '{}+{}は{}です'.format(1,1,2)
であれば、3つのリテラルがあること、そのリテラルを組み合わせた文字列を生成したいことがわかりやすい。
このように「人間にとっての読みやすさ（リーダブルコード, readable code）」の観点から、str.format()形式が提供されている。



前述スライドの str.format() の例示。

プログラミング1

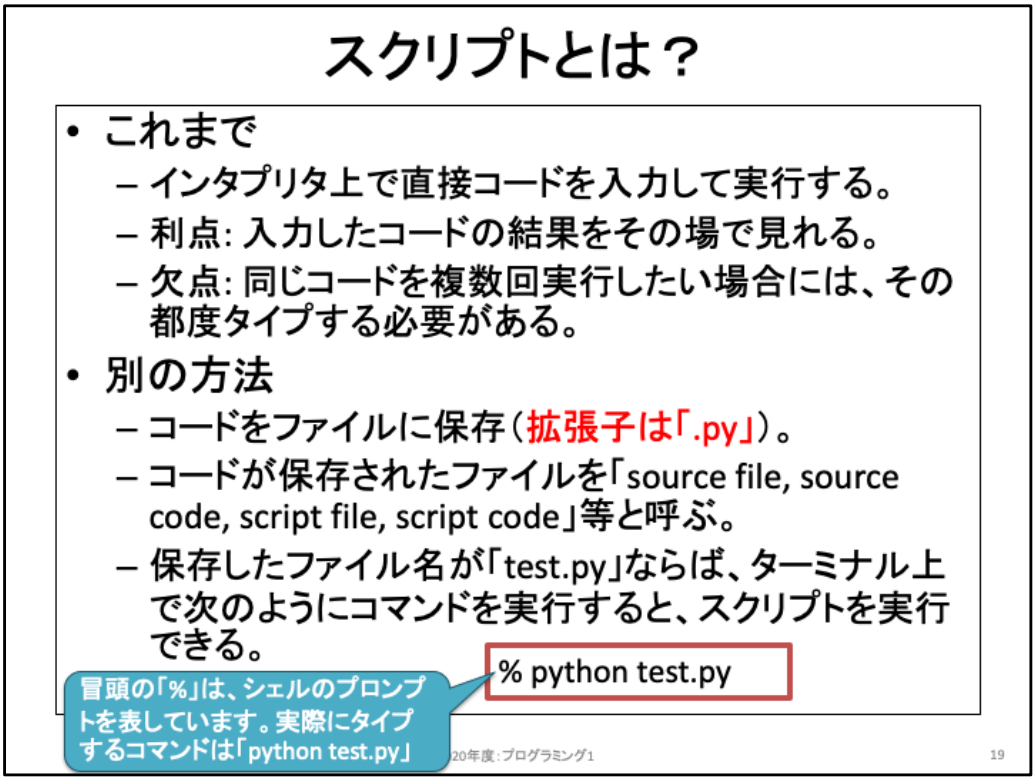
(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

基本演算とstr.format書式を読めるようになろう。

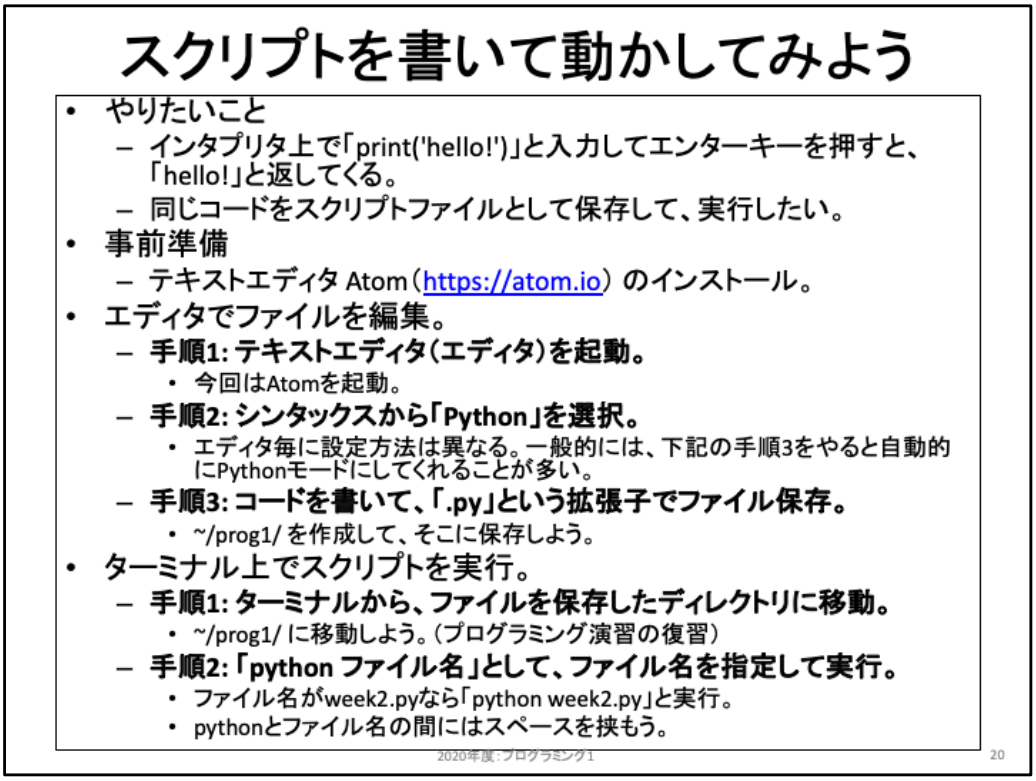
スクリプトの利用
（教科書に書かれてません）

スクリプトとは？
スクリプトを書いて動かしてみよう
スクリプト vs.インタプリタ



ここまでは、Pythonインタプリタを起動し、そこでコードを直接書くことで命令をコンピュータに伝え、実行させていた。しかしこの方法では、同じ命令をさせたい場合にはその都度同じコードを書く必要があり、手間である。この手間を省くため、コードをテキストファイルに保存して実行する方法がある。この「コードが保存されたファイル」のことを「スクリプト」と呼んでいる。

スクリプトとしてコードを保存する際には、テキストエディタを使おう。テキストエディタとは、テキストのみを編集（edit）してファイル保存するためのソフトである。プログラミングにおいては「プログラミング言語仕様に則ったコード」をそのままテキストとして保存する必要があるため、テキストに特化したソフトを使う。テキストエディタには色んな種類があり、それぞれ癖があるので、いろいろ試してみることで使いやすいものを後日探してみるといいだろう。



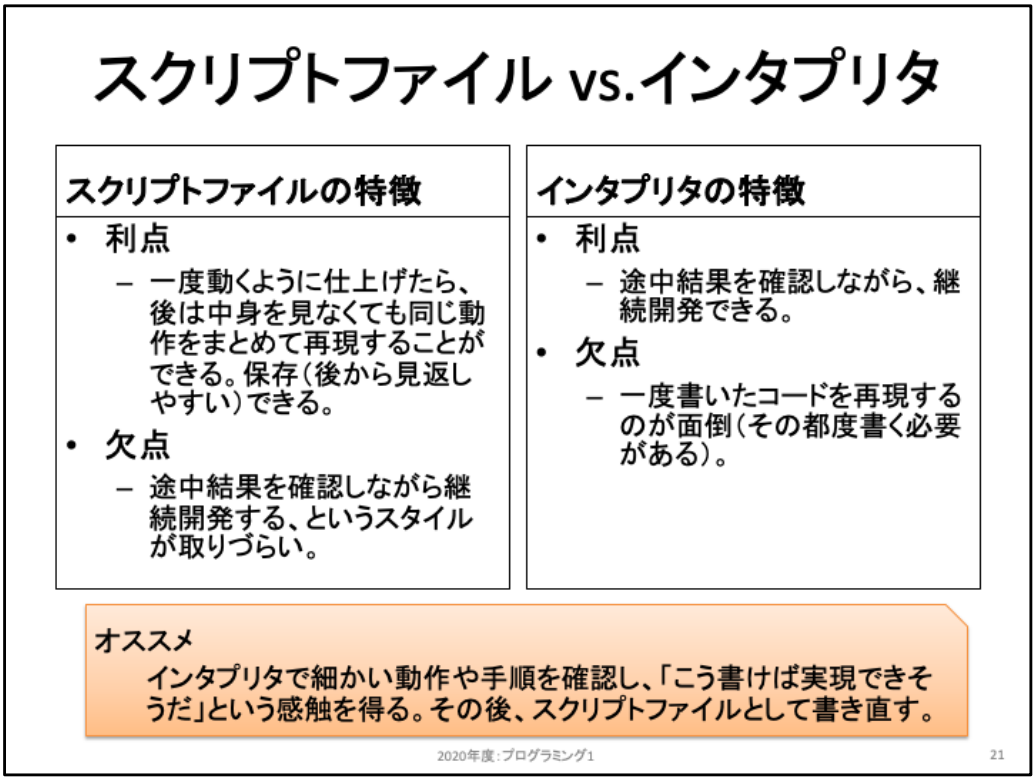
今回は Atom https://atom.io をインストールして、使ってみよう。Downloadからソフトウェアをダウンロードすると、atom-mac.zipが ~/Downloads/ 以下に保存される。この ~/ は、「実行したユーザのホームディレクトリ」という意味である。あなたが使っているPCは、初めてセットアップした際にあなた専用のアカウントを作ったはずだ。これは複数のアカウントを作ることが出来る（例えば家族で共用して使うとか）が、その際に各アカウント毎に「ここは私専用のフォルダ」「あそこはあなた専用のフォルダ」といったことを決めておいた方が使いやすい。各自のフォルダをホームディレクトリと呼び、先程は ~/ で表示した。このフォルダやディレクトリとはOS毎に呼び名が変わるだけで、基本的には同一のものと考えておこう。

Finderを起動すると自動的にホームディレクトリを表示してくれる。この中にある「ダウンロード」というフォルダは、実際には「Downloads」という名前であり、OSの言語指定で日本語を指定している場合に自動で翻訳されているだけである。Finderからダウンロードフォルダを開くと、先程のatom-mac.zipが見つかるはずだ。これをダブルクリックするとAtom.appというアプリケーションが展開されるので、これを「アプリケーション」フォルダにドラッグ・アンド・ドロップで移動（＝アプリケーションとしてコピー）しよう。これ以降はアプリケーションフォルダにあるAtom.appをダブルクリックすると起動することができる。

Atomを起動した直後はWelcome, Welcome Guide, untitledの3つのタブが開いていると思われる。細かいドキュメント等は一先ず置いておき、ここではコードを保存して実行するまでの流れを覚えよう。まず(1)untitledタブをクリックする。(2)そこにPythonコードを書き並べる。(3)書き終えたら、今回は「test.py」という名前で保存しよう。実際にはファイル名は変数名として利用できるものなら何でも構わないが、拡張子は必ず「.py」とすること。またファイルを保存するディレクトリは、自分が覚えられる場所ならどこでも構わない。プログラミング1用のディレクトリを作成し、そこに保存するようにすると他授業のファイルと区別しやすいが、各自がやりやすい形で保存して構わない。例えばホームディレクトリにtest.pyという名前で保存したのならば、そのファイルの場所は ~/test.py となる。ホームディレクトリの下に prog1 というディレクトリを作成し、その中に week2.py という名前で保存したのならば、~/prog/week2.py として保存されている。

ファイルを保存し終えたら、(3)ターミナルを起動し、保存したディレクトリに移動しよう。ホームディレクトリに保存したのであれば、ターミナルを起動した直後にその場所にいる。ファイルの有無を確認するためには、シェル上で ls コマンド（一覧listの省略）を入力し、実行してみよう。想定通りならばファイルが見つかるはずだ。ホームディレクトリと異なるディレクトリ、例えば前述のように ~/programming1 というディレクトリに保存したのであれば、cd コマンド（change directoryの省略）を使って移動しよう。具体的には、シェル上で cd programming1 と実行するとそこに移動できる。移動したあとは同様に ls コマンドでファイルを確認しよう。ファイルが見つかったならば、(4)「python ファイル名」として実行してみよう。ちゃんとファイルがあり、中に書かれているコードが適切ならば、想定通りの動作結果が出力されるはずだ。ls, cd 等はUNIXコマンドとも呼ばれており、詳細はプログラミング演習1で扱う。

これで、ファイルとして保存したプログラムを実行できるようになった。何度でも再実行することができるし、一部分を修正して利用する（＝再利用する）こともしやすくなったはずだ。



慣れてくると最初からテキストエディタを使った開発が早いが、言語仕様に慣れていない段階ではいろいろと細かく試しながら確認できるインタプリタの方が便利なことも多い。例えばAtomとターミナルの2つを立ち上げておき、インタプリタで納得できるコードが書けたらそれをエディタにコピーして保存するとか、双方の利点を使い分けるといいだろう。

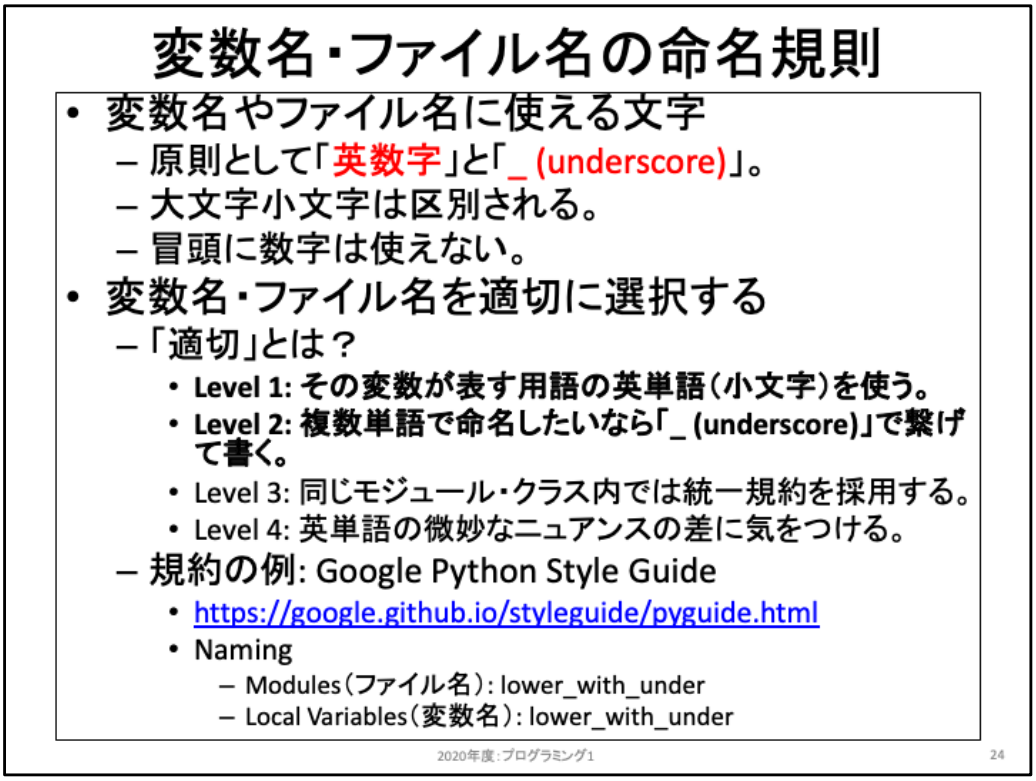
プログラミング1

(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

インタプリタ実行とファイル実行を使い分けよう。

変数名・ファイル名の命名規則
（教科書に書かれてません）



変数名やファイル名として使うことの出来る文字と、規約を覚えておこう。
冒頭3項目はPythonの言語仕様としてのルールであり、厳守する必要がある。

その次の Level3～Level4、Google Python Style Guide は規約の例、言い換えるとローカルルールである。言語仕様を守っただけの変数名では分かりにくい名称が使われることも多く、読みにくいコードになることがある。例えば四角形の面積を求めるために、

```
>>> area = height * width
```

と、

```
>>> a = b * c
```

とではどちらがコードの意図を汲み取りやすいだろうか。意図の汲み取りやすいコードは読みやすいコードとなっており、その分バグを発見しやすくなるため保守しやすい。現場ではコードを一度書いて終わりではなく、何度も修正・機能追加等を伴う編集をしながら使い続けていくため、読みやすさを意識したコードが望ましい。そのためにも組織毎にローカルルールを定め、それに則る名称を利用することが一般的である。多くの場合に共通するLevel2ぐらいまでを意識して名付けるようにしてみよう。

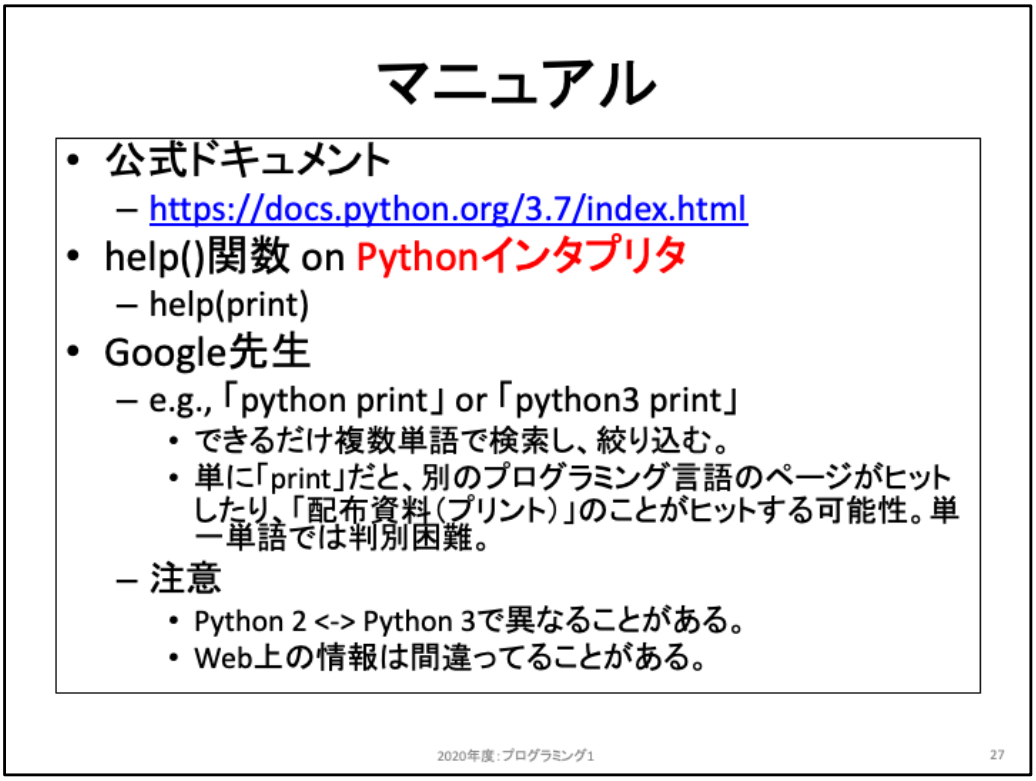
プログラミング1

(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

慣習を守ることで「他人が読みやすいコード(readable code)」になる。

マニュアルの参照
（教科書に書かれてない
or
後回しになってます）



関数に関するドキュメントは、上記URL先から探すか、直接Pythonインタプリタ上でhelp()関数を使うと参照することができる。例えばprint()関数について調べると以下のような出力が得られるはずだ。

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='¥n', file=sys.stdout, flush=False)
    （中略）
(END)
```

print(value, ..., sep=' ', 略)は使い方を簡易的に示している。ENDでマニュアルの最後であり、マニュアル画面から抜け出してインタプリタに戻りたい場合には、q（quitの頭文字）を入力しよう。

valueはリテラルや変数を記述する欄であり、それらをカンマ（,）で区切って列挙できることを「value, ...,」として示している。

その次の sep=' ' は、数行下に説明があり「string inserted between values, default a space.」とある。これは、value欄に複数の値を列挙した場合、それらの値を何らかの文字を挿入することをしめしている。例えば、

```
>>> print(1,2,3)
```

の結果を確認してみよう。自動的にスペースが挿入された状態で指定した値が出力されるはずだ。このように「特に指定しなかった場合に自動で使われる値」のことを「デフォルト値」と呼ぶ。今回の場合にはスペースがデフォルト値として指定されており、これを変更するには以下のように指定する。

```
>>> print(1,2,3,sep='#')
```

end は、出力後に自動で追加される制御文字である。ここでは円マークで掲載しているがこれはPowerPointの都合であり、正しくはバックスラッシュ（backslash）だ。入力する際にはバックスペースの左隣にある円マークの付いたキーを押すと、バックスラッシュになるはずだが、もしならない場合には「Optionキーを押しながら円マークキーを押す」ことでバックスラッシュが入力できるはずだ。'バックスラッシュn'はこれで1文字の制御文字であり、「改行文字」を意味する。改行文字をsepとして設定するとどうなるだろうか。試してみよう。

Flush は通常無視して構わない。補足しておくとこれはコードの実行速度も考慮した話であり、四則演算と比べると「標準出力に書き出す」という処理はとても遅い。このため、print文が大量にあると全体として処理が遅くなってしまうが、通常はなるべくその負荷が小さくなるような工夫がなされている。逆にこの工夫のために、出力し終える前に別のコードの影響でプログラム全体が異常終了してしまうことがある。このような状況を避けるため、必ず出力し終えてから次に進むように指定するのがflushである。

プログラミング1

(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

help()やオンラインマニュアルを活用しよう。

従来は「二人で一台のPCを使ってプログラミングする」ペアプログラミングを実施していましたが、残念なことに今年度は当面なしにします。

ペアプロ演習

- 授業ページを参照

宿題

- 復習: 適宜
- 課題レポート1: ∝切: webページ参照
- 予習: 教科書読み
  - 2章
    - 2.3 Strings and Input
    - 2.4 Iteration
  - (スキップ) 3章
  - 4章
    - (スキップ)4章冒頭
    - 4.1.1 Function Definitions
- 復習・予習(オススメ): progate

プログラミング1

(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
    1. Calculations and Remembers
    2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
    1. Glossaries, 用語集1, 2, 3
    2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
    1. スクリプトとは？
    2. スクリプトを書いて動かしてみよう
    3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

欲しい出力を得るためのレシピを考える必要がある。レシピ≒アルゴリズム。

レシピを記述するための道具（基本的な型・算術演算子・比較演算子・論理演算子）を使えるようになろう。

基本演算とstr.format書式を読めるようになろう。

インタプリタ実行とファイル実行を使い分けよう。

慣習を守ることで「他人が読みやすいコード（readable code）」になる。

help()やオンラインマニュアルを活用しよう。

参考文献

- 教科書: Introduction to Computation and Programming Using Python, Revised And Expanded Edition
- Python 3.7.3 documentation, https://docs.python.org/3.7/index.html
- Google Python Style Guide, https://google.github.io/styleguide/pyguide.html