cs-11. Python プログラミング応用

（コンピューターサイエンス）

金子邦彦

資料中の図などは Google Nano Banana 2 を用いて作成

「コンピュータサイエンス」第１１回の内容

プログラミングでの抽象化は、共通点のある処理を１つにまとめることである

同じ形の処理が並んでいる

- 100 × 1.1
- 150 × 1.1
- 400 × 1.1

共通部分を1つにまとめる

a は変わる部分

a × 1.1

関数 と 引数

関数は，処理に名前を付けたもの

```
def kakeru(a):
    return a * 1.1
```

中で使う a は仮引数（パラメータ）

100
150
400

呼び出し時に入る
具体的な値
100，150，400 は実引数

抽象化前：同じ修正を何か所も直す

抽象化後：1か所を直せばよい

修正漏れが起こりにくくなり，変更時の誤りを減らせる

はじめに

この資料の流れ：具体から始め、抽象化で整理し、エラーを手がかりに意図を正す

資料の流れ（4ステップ）

① 具体的な計算を書く
100 × 1.1 など

→

② 共通点を見つけ、変わる部分を変数にする
a × 1.1（抽象化の土台）

→

③ 名前を付けて関数にまとめる
再利用できる（呼び出して使う）

→

④ 意図とのずれはエラーが教える
エラー＝意図と動作のずれの手がかり

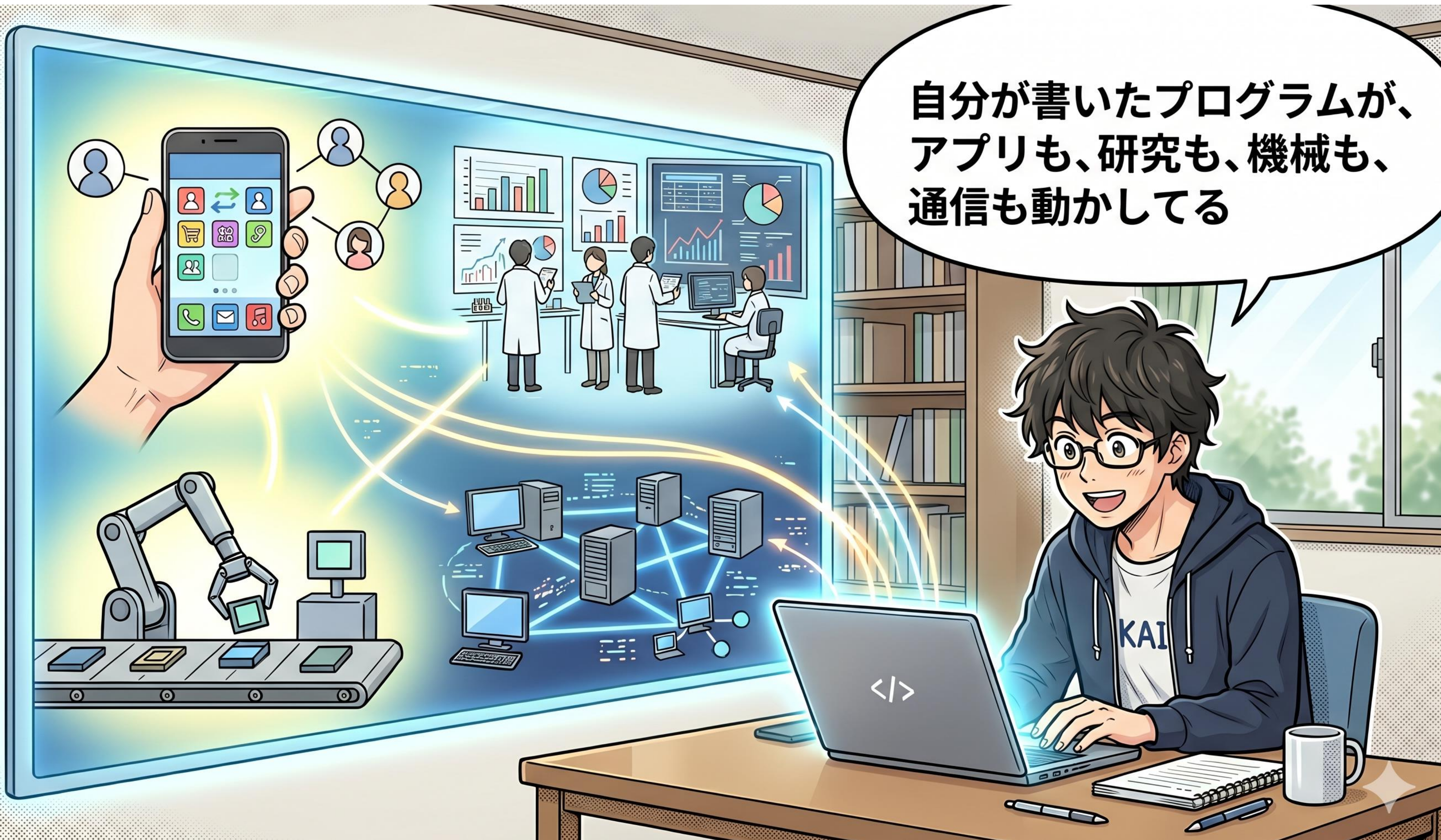
抽象化の定義

抽象化とは

共通部分を取り出し、変わる部分は後から変えられる形（変数・仮引数）で残すこと

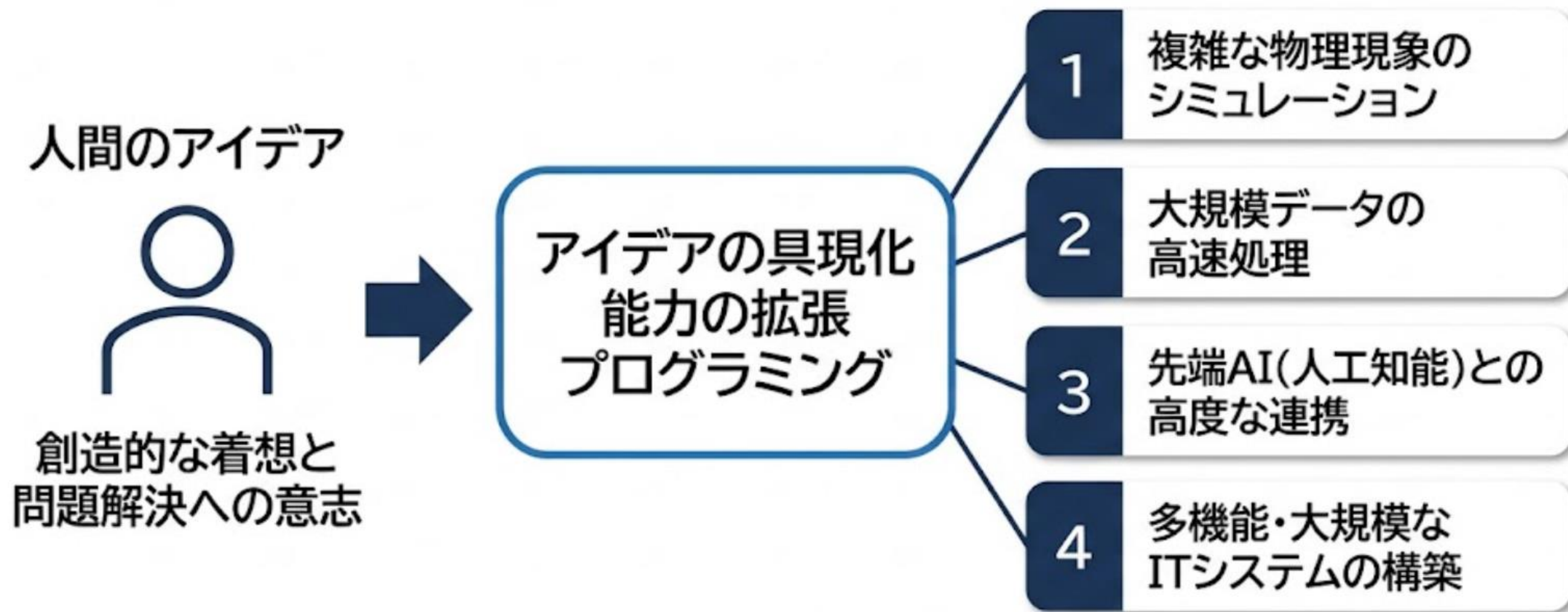
変数でまとめる ＝ 土台 ＋ 名前と再利用 ＝ 関数

変数でまとめるのが土台、名前と再利用を加えたものが関数



本授業の図は生成AIである Google Nano Banana 2 を用いて作成

プログラミングで何ができるのか



プログラムとは — 命令を書いた手順の集まり

指示を与える

```
a = [200, 400, 300]
for i in a:
    print(i * 1.1)
```

自動で処理

200 / 400 / 300

結果を得る

220.0 / 440.0 / 330.0

だから複雑な作業も【自動化・効率化】できる

エラーの意味と対処

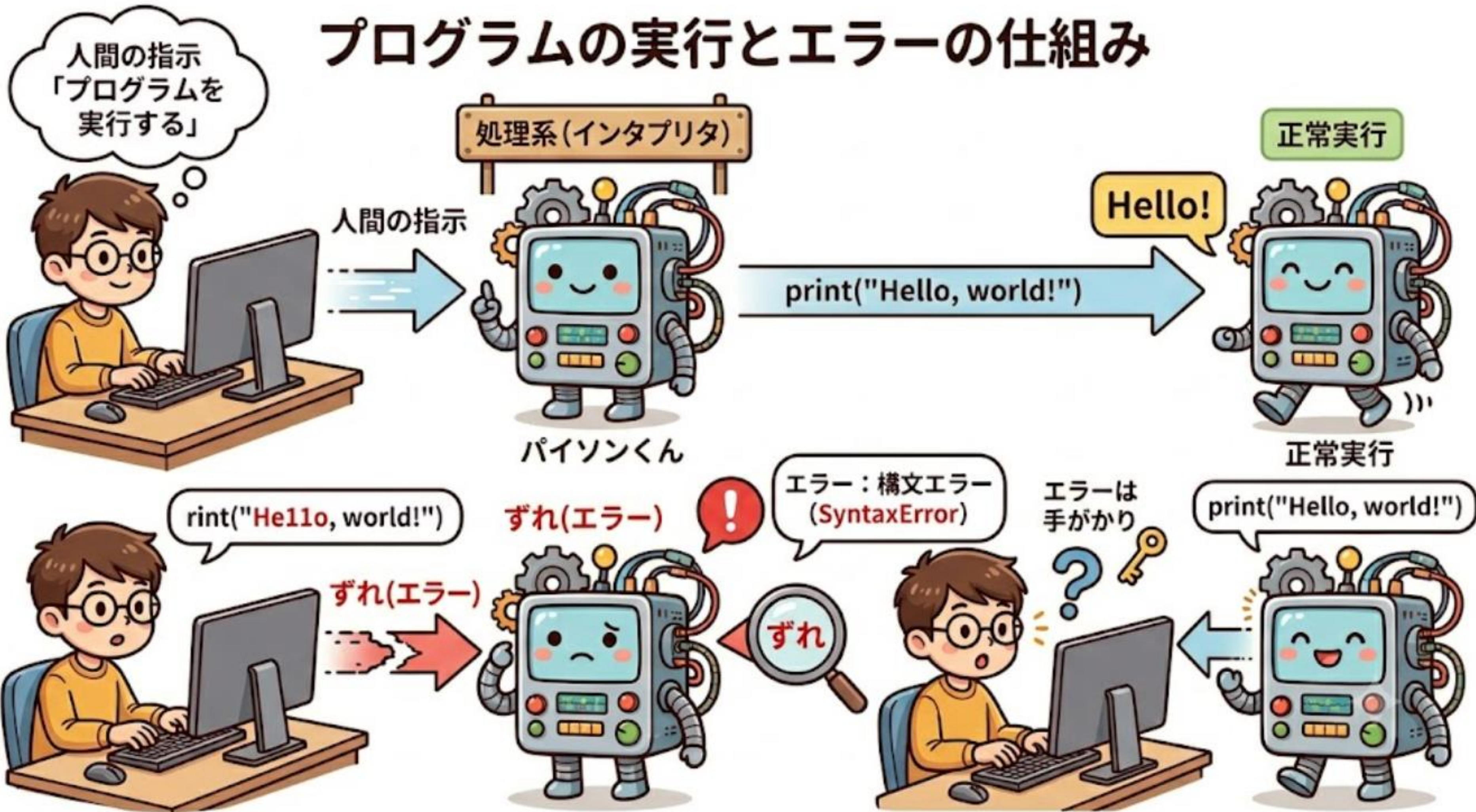
エラーは故障じゃない。直す場所を教えてくれる『手がかり』だ

① 最終行を読む　② 直前の変更を戻す　③ printで値を確認　④ 検索・AIに尋ねる

Python のエラー

Database Lab.

Database Lab.



エラーが出ても落ち着いて

1

エラー文の
最終行を読む

エラーの種類と
内容が書かれている

```
NameError: name 'x'
 is not defined
```
= 名前が定義されていない

2

直前の変更を
元に戻す

戻してエラーが
消えるか確かめる

3

print() で
途中の値を確認

意図した値が
入っているか調べる

```
print(x))
```

4

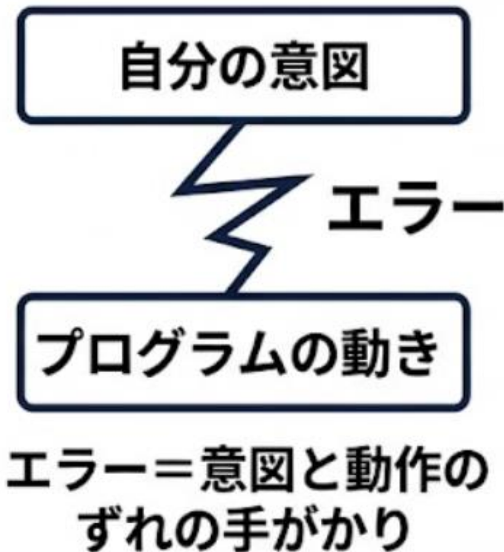
検索・AIに
尋ねる

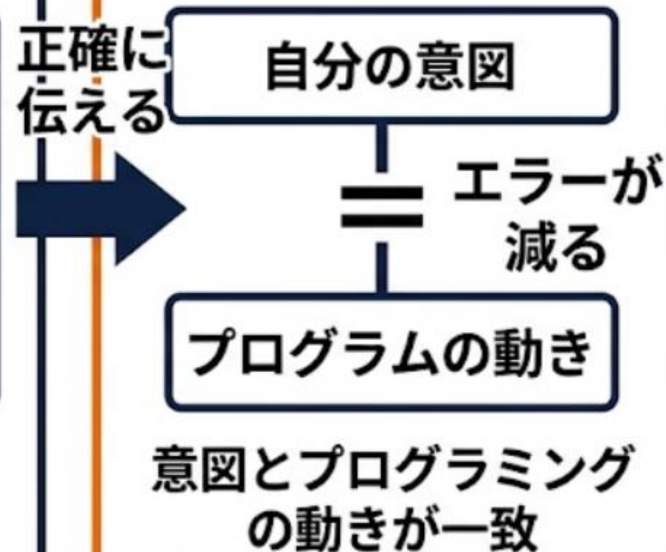
エラー文とコード
を調べる、または
AIに質問する

ただしAIや検索の答えが正しいとは限らない。必ず自分で動かして確かめる

エラーは、意図とプログラムのずれ

エラーは失敗ではなく『意図とプログラムのずれ』を示す手がかり





抽象化で整理 → 意図を正確に伝える ＝ プログラミング学習の基礎

式の抽象化と関数

Database Lab.

同じような計算は『共通部分』をまとめると、間違いが減る

また同じの
書くの面倒…

× 1.1 は共通だ！
変わるのは数字だけ

α × 1.1

α に 100・150・400 を入れる

共通部分をまとめる = 抽象化。
直す場所も１か所で済む

抽象化

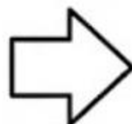
抽象化＝共通点のある処理を1つにまとめること

バラバラの計算

100 × 1.1
150 × 1.1
400 × 1.1

似た計算がいくつもある

共通点を見つける

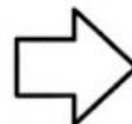


共通部分に気づく

100 × 1.1
150 × 1.1
400 × 1.1

変わる部分（値）と共通部分（×1.1）を分ける

変数でまとめる



変数を使った式

a × 1.1

a に 100・150・400 などが入る

1つの式で全部を表せる

```
for a in [100, 150, 400]:
    print(a * 1.1)
```

a × 1.1 を繰り返し（または関数）の中で1回だけ書く。だから直すのは1か所で済む

バラバラの計算 → 共通部分を見つける → 変数でまとめる ＝ 抽象化

抽象化

抽象化とは、共通部分を取り出し、変わる部分は変えられる形で残すこと

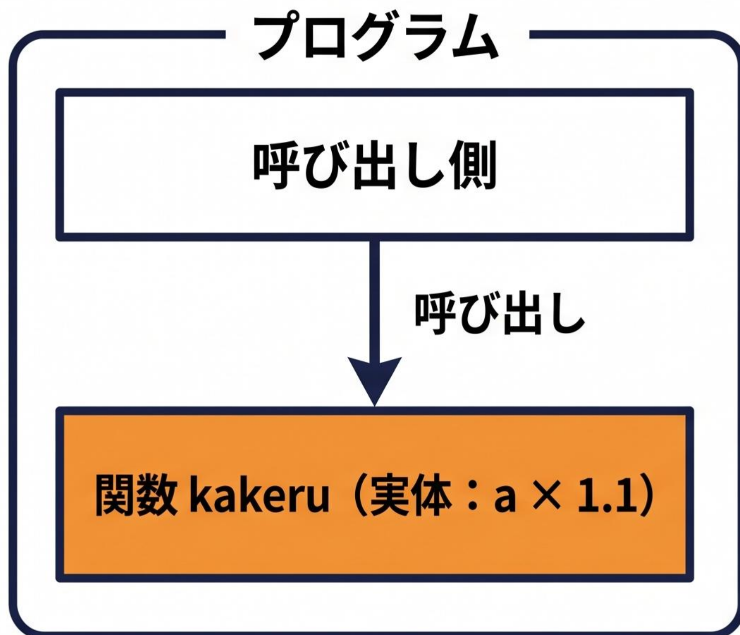
① **共通する部分**を取り出す
② 変わる部分は『**あとで変えられる場所**』にする

具体	抽象
100 × 1.1 150 × 1.1 400 × 1.1 変わる部分（濃紺）と 共通部分（オレンジ）	a × 1.1 変えられる場所（100・150・400が入る） いつも同じ（共通部分）

ばらばらの具体 → 共通部分（× 1.1）はそのまま、変わる部分（値）は変数 a に = 抽象化

抽象化と関数

関数＝値を入れると結果が返る。



呼び出し側が値を渡して、関数を使う

同じ式 a × 1.1 に、違う値を入れるだけ

値を入れる（100・150・400）→ 同じ式 a × 1.1 で処理 → 結果が返る（110・165・440）

数学の関数とプログラムの関数の違い

数学の関数は『対応関係』、プログラミングの関数は『呼び出して実行する手続き』

同じ『関数』でも意味が違う

数学の関数

$$f(x) = x + 1$$

入力に対して出力が一意に定まる対応関係
書いた時点で値が決まる（実行という考えがない）

2 → 3

プログラミングの関数

```
def kakeru(a):
    return a * 1.1
```

定義しただけでは実行されない
呼び出して初めて実行され、返り値が返る

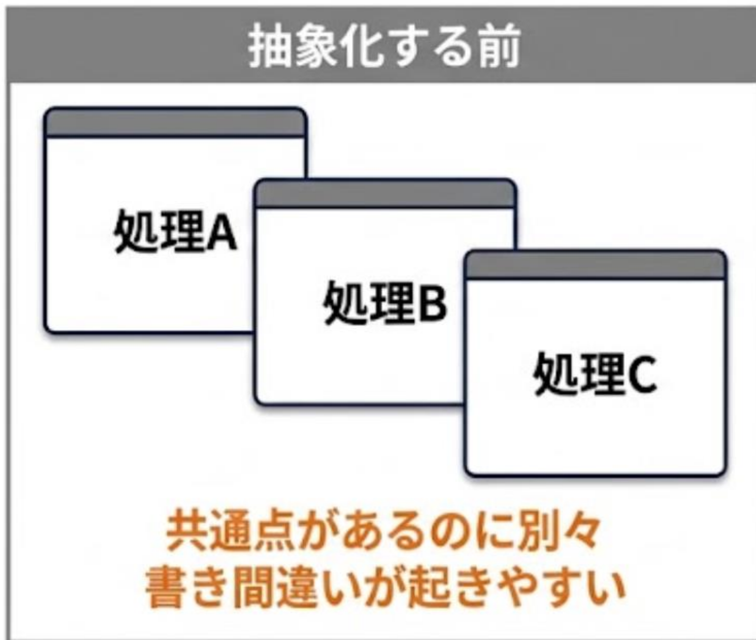
```
kakeru(100) → 110
kakeru(150) → 165
kakeru(400) → 440
```

数学＝静的な対応、プログラミング＝呼び出して動く手続き

抽象化の効果

抽象化＝共通点のある処理から共通部分を取り出し、名前を付けて1つにまとめること。
意図が整理され、間違いが減る。

Before / After



抽象化

意図が整理される → 間違いが減る（エラーが出にくくなる） → AIツールへの指示にも生きる

抽象化の効果

Database Lab.

直す場所が1か所だから、間違いが減る

抽象化しないと

抽象化すると

```
def kakeru(a):
    return a * 1.1
```

ここを直すだけ

直す場所が3か所 → 書き間違いが起きやすい

直す場所は1か所 → 間違いが減る

3か所直す → 1か所だけ直す ＝ 抽象化の効果（間違いが減る）

関数定義

DataBase Lab.

マシンを自分で組み立てる
＝関数を定義する

カードを入れる
（実引数）

ボール作成マシン（関数）

```
def make_ball(color):
    return color + "のボール"
```

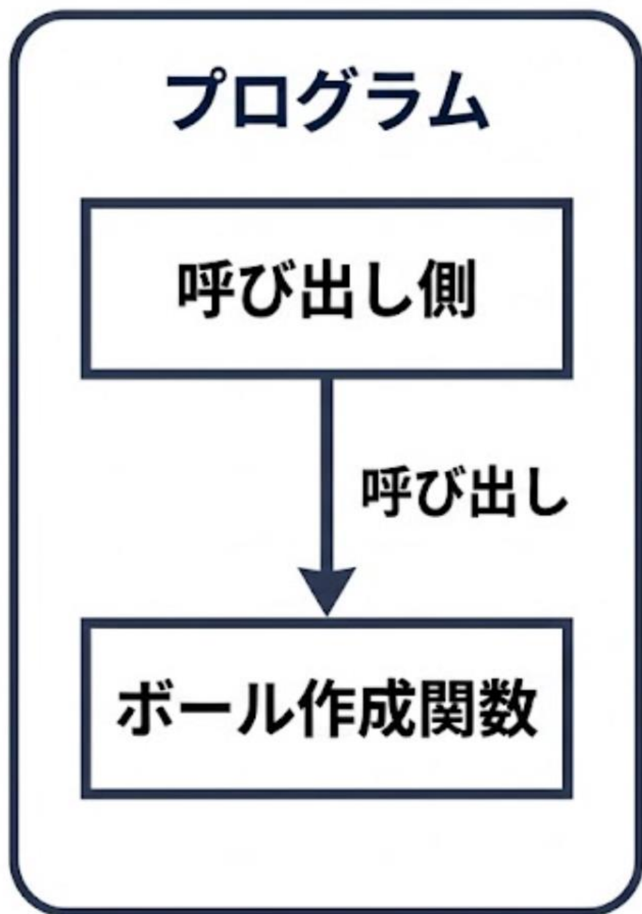
同じ色の
ボールが出る
（戻り値）

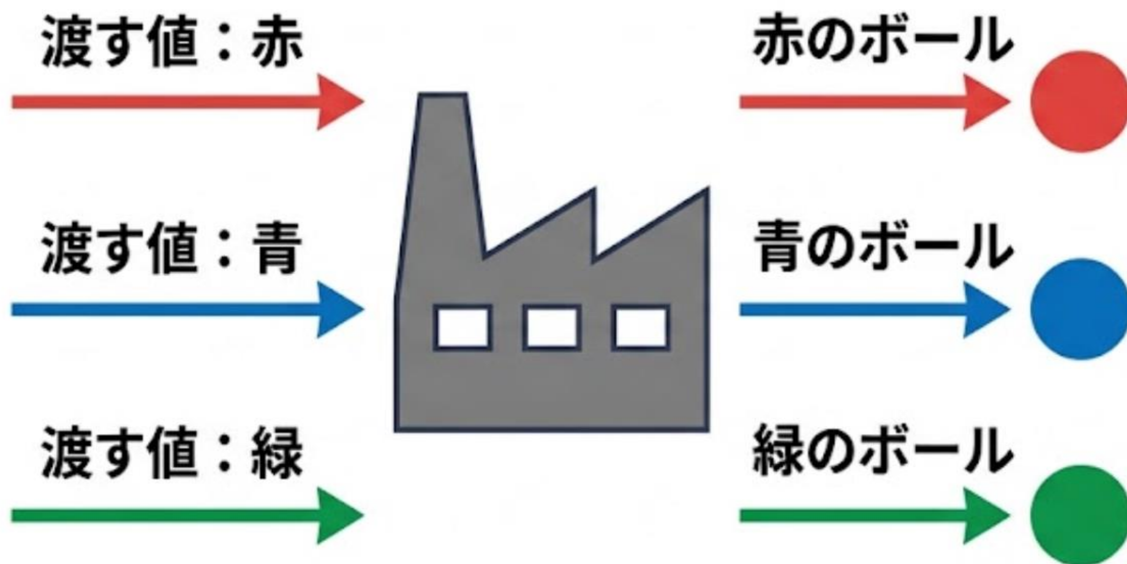
一度作れば、入れる値を変えるだけで何度でも使える

関数

Database Lab.

関数に値を渡すと、それに応じた戻り値が出てくる





同じ関数でも、渡す値を変えると戻り値が変わる

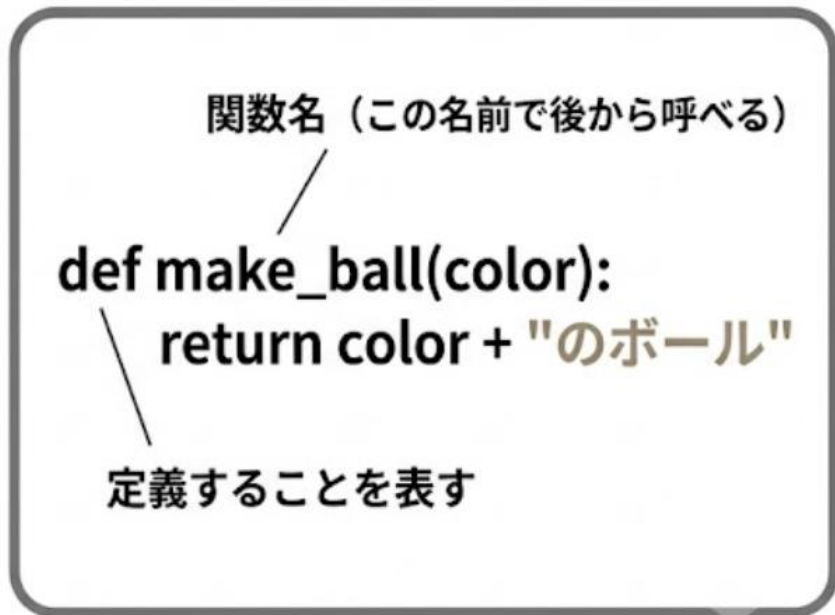
関数定義

関数定義とは、一連の処理に名前を付けて1つにまとめ、後で使えるようにすること

名前のないバラバラの処理

名前を付けて
まとめる

関数の定義



関数定義

関数定義とは、名札（関数名）を関数の実体（処理の本体）に結びつけること

```
def make_ball(color):
```

結びつける宣言

make_ball

関数名（名札）

『def make_ball(color):』
と書くと、make_ball という
名札が関数の実体に結びつく

```
make_ball(color):
    return color + "のボール"
```

関数の実体（処理の本体）

同じ名札（make_ball）で、
この実体を後から何度でも呼び出せる

関数における抽象化

変わる部分を仮引数 color に置き換えると、3 つの文字列が 1 つの式にまとまる

3 つの具体例

"赤のボール"

"青のボール"

"緑のボール"

変わる部分を名前に
まとめる（抽象化）

仮引数（変わる部分の置き場所）

color + "のボール"

共通部分

color に 赤・青・緑 が入ると、
3 つの具体例に戻る

実引数、仮引数（パラメータ）、戻り値

仮引数＝関数側であらかじめ決めた受け皿（値を受け取る名前。1つまたは複数)。
実引数＝呼び出すときに実際に渡す値（呼び出すたびに変わる)。

実引数
(呼び出しごとに変わる入力)

関数のしくみ

戻り値（出力)

実引数：赤

実引数：青

実引数：緑

関数名 (例：ボール作成関数)

仮引数
(パラメータ)

関数の実体
(処理の本体)

戻り値：
赤のボール

戻り値：
青のボール

戻り値：
緑のボール

実引数、仮引数（パラメータ）、戻り値

実引数を渡すと仮引数で受け取り、関数が処理して戻り値が返る

関数の定義

関数名

仮引数（パラメータ）：
変数1つまたは複数

```
def make_ball(color):
    return color + "のボール"
```

戻り値

関数の呼び出しと戻り値

関数の呼び出し	戻り値
make_ball("赤") →	"赤のボール"
make_ball("青") →	"青のボール"
make_ball("緑") →	"緑のボール"

実引数：呼び出すたびに変わる（渡す値）

渡す実引数を変えると戻り値が変わる

関数定義と呼び出し

定義は1回だけ。呼び出すたびに、その定義済みの関数が実行される

関数の呼び出し（何度でもできる）

関数の定義（1回だけ書く）

```
def make_ball(color):
    return color + "のボール"
```

```
make_ball("赤")
make_ball("青")
make_ball("緑")
```

呼び出して初めて実行される

関数と呼び出し

ベタ書き3行と、関数を定義して呼ぶ書き方は、同じ結果になる

ベタ書き（同じような行を3回書く）

```
print("赤のボール")
print("青のボール")
print("緑のボール")
```

＝

画面表示は
まったく同じ

関数を定義して呼ぶ

関数名　仮引数

```
def make_ball(color):
    return color + "のボール"

print(make_ball("赤"))
print(make_ball("青"))
print(make_ball("緑"))
```

返り値

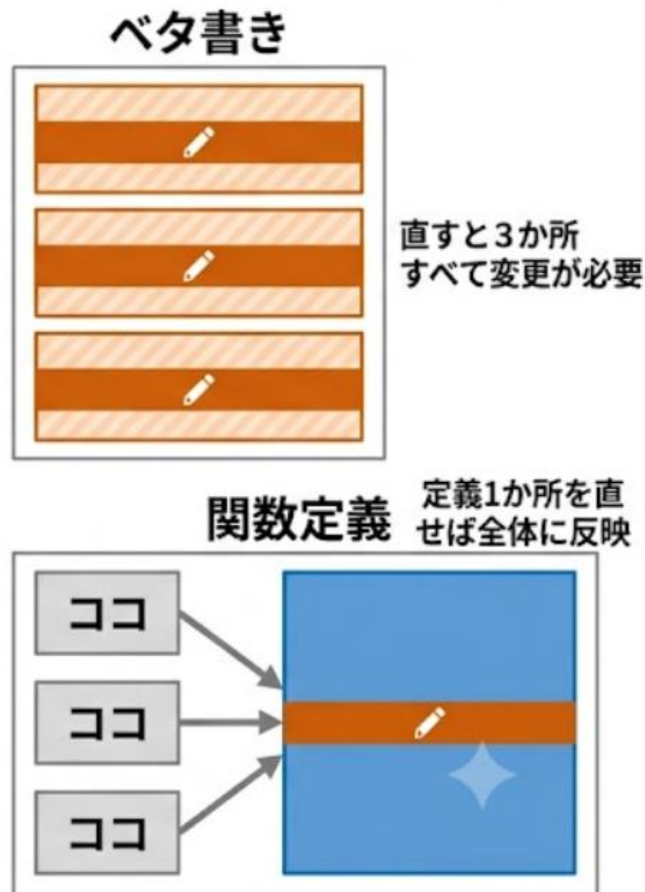
実引数

関数にすると、共通の処理を
１回書くだけで、何度でも使える

関数の有用性

同じ処理を何度も使うなら関数定義が有利。1回限りの短い処理ならベタ書きでもよい

観点	ベタ書き	関数定義
コードの重複	同じ処理を何度も書く	1回だけ書いて再利用
修正のしやすさ（保守性）	全箇所を直す必要がある	定義1か所を直せばよい
再利用性	低い（その場限り）	高い（何度でも呼べる）
変更時のバグ	直し忘れが起きやすい	起きにくい
向いている場面	1回限り・ごく短い処理	同じ処理を繰り返す・長い処理



コードの再利用性

Database Lab.

税込価格の計算を1回だけ定義しておけば、どの画面でも呼ぶだけで使える（再利用性）

まず定義する

```
def zeikomi(price):
    return price * 1.1
```

税込価格を計算する処理を1回だけ書く

1

商品一覧の画面で使う

```
print(zeikomi(100))
→ 110.0
```

定義した関数を呼ぶだけ

2

後日、別のレシート画面画面でも使う

```
print(zeikomi(250))
→ 275.0
```

同じ計算を書き直さず、呼ぶだけ

3

定義は1回、呼び出しは何か所でも

```
zeikomi(100)
zeikomi(250)
zeikomi(980)
```

新しい画面が増えても、書くのは呼び出し1行だけ

4

コード修正時のバグ防止

税率の計算を複数箇所にベタ書きすると、変更時に直し忘れが起き、一部だけ古い税率のまま残る（変更時のバグ）

ベタ書き：税率 10%が 3 か所

```
print(100 * 1.1)
print(250 * 1.1)
print(980 * 1.1)
```

1

同じ 1.1 が 3 か所に散らばる

税率が 8% に変更。全部直す必要

```
print(100 * 1.08)
print(250 * 1.08)
print(980 * 1.08)
```

2

3 か所すべて 1.08 に直す必要がある

1 か所を直し忘れる＝バグ

```
print(100 * 1.08)
print(250 * 1.08)
print(980 * 1.1)
```

3

ここだけ古い税率のまま＝バグ

関数なら定義 1 か所だけ

```
def zeikomi(price):
    return price * 1.08

print(zeikomi(100))
print(zeikomi(250))
print(zeikomi(980))
```

4

定義 1 か所を直せば全部正しく変わる＝直し忘れが起きない

Database Lab.

演習

演習１

①trinketの次のページを開く

https://trinket.io/python/68a090babf

② 実行結果が，次のように表示されることを確認

main.py

```
def foo(a):
    return a * 1.1
print(foo(100))
print(foo(150))
print(foo(400))
```

Powered by

```
110.0
165.0
440.0
```

演習２から６の内容

お絵かき用の turtle が、数十行で『自分で操作して遊べるゲーム』になる

1. 絵を描くだけの turtle がゲームになる

標準ライブラリだけで作れる

2. キーを押すとキャラが本当に動く

自分の操作が画面に伝わる手応え

3. 受け止めた瞬間スコアが増える『できた！』

成功が数字で返ってくる達成感

4. たった数十行で動く・操作・得点が成立

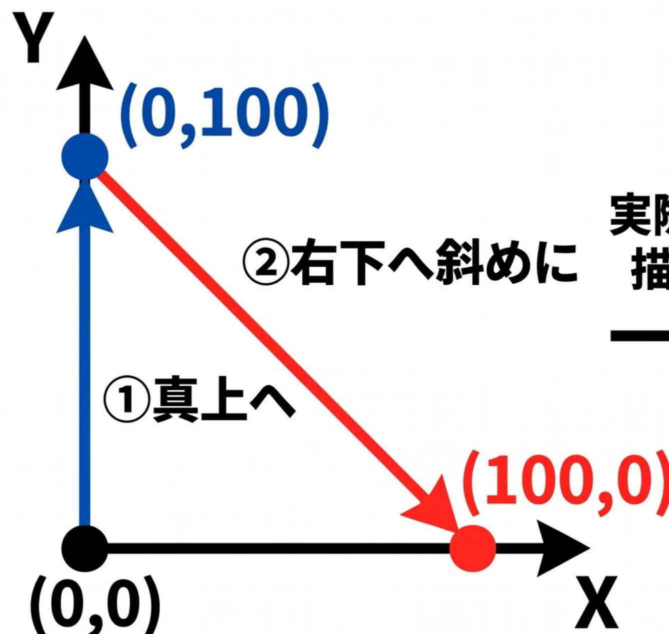
座標と乱数が画面の動きに変わる

タートルグラフィックス

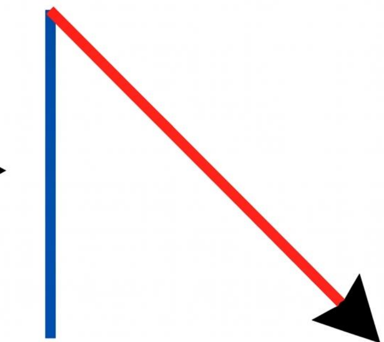
タートル（亀; turtle）が線を引きながら進む

```
import turtle
t = turtle.Turtle()
t.goto(0,100)
t.goto(100,0)
```

《Python プログラム》

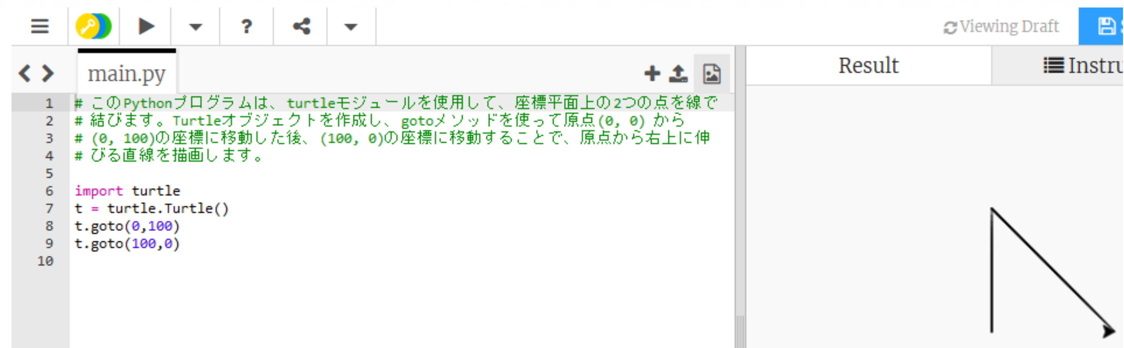


実際の描画



描かれる図形

goto(x,y)：現在地から座標(x,y)まで線を引いて移動する



Goto メソッドでの場所の指定法

タートル（亀）の出発点は（0, 0)。そこを基準に上下左右へ位置が決まる

Y

上へ ＋（プラス）

(0, 100)

(0, 0) スタート位置

X

左へ ー（マイナス）

右へ ＋（プラス）

(100, 0)

下へ ー（マイナス）

(0, 100)

原点から上へ

(100, 0)

原点から右へ

タートルグラフィックスのオブジェクトとメソッド

タートルの分身 t に命令（メソッド）を出して絵を描く

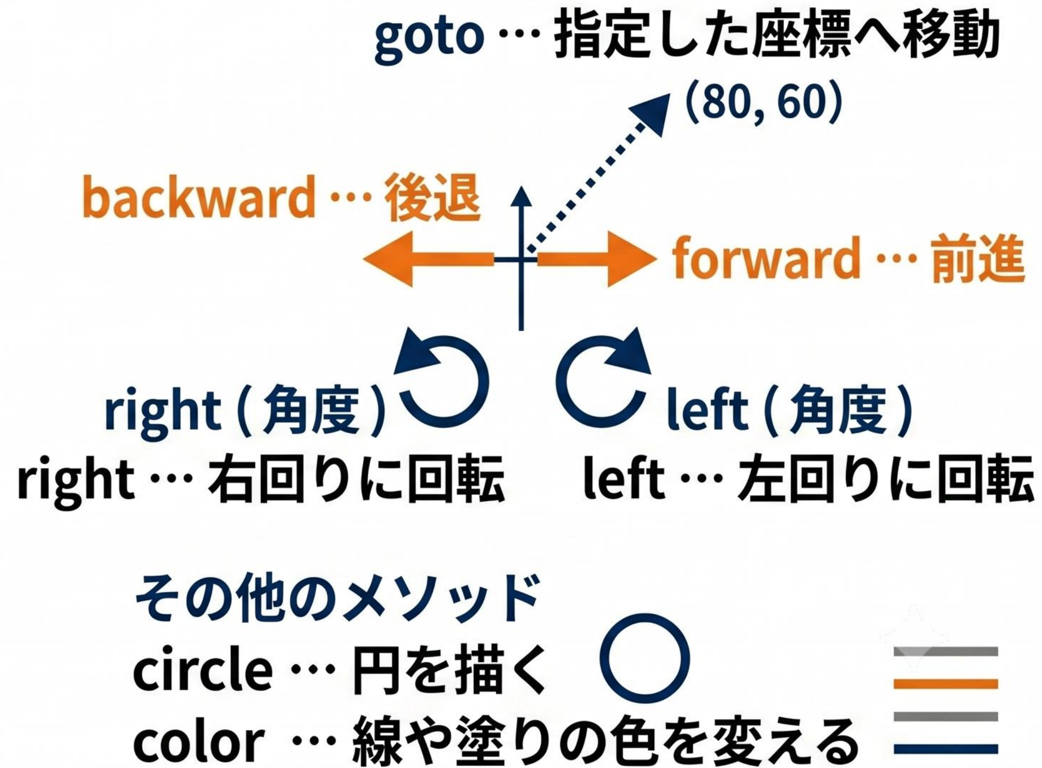
オブジェクトとメソッドの関係

オブジェクトメソッドの関係

```
t = turtle.Turtle()
t.goto(0,100)
t.forward(50)
```

t に対して点（.）でメソッドを呼ぶ

主要メソッドと移動対応



演習２，３，４，５，６の流れ

- Python 標準ライブラリの turtle で「**落下物キャッチ(受け止め)ゲーム**」を段階的に作ります。
- **完成形(演習6)**は、**左右キーで動かす四角形のプレイヤー**が、**上から落ちてくる落下物を受け止めて得点するゲーム**です。

使用する turtle 機能

グラフィックス機能

- **Screen()**:描画ウィンドウを取得・作成する
- **setup(幅, 高さ)**:ウィンドウの幅と高さを指定する
- **penup()**:ペンを上げて、以後の移動で線(軌跡)を描かないようにする。タートルを作った直後・最初の goto より前に呼ぶと、初期移動の線も残らない
- **goto(x, y)**:指定した座標へ一気に移動する
- **setx(x)**:x 座標だけを変えて移動する
- **xcor()**:現在の x 座標を取得する

使用する turtle 機能

キーボード検知

- **screen.listen()**:キー入力の受け付けを開始する。**これを呼ばないと onkey の登録があってもキーを受け取れない**ので、キー設定の最後に必ず一度呼ぶ。
- **screen.onkey(関数, "キー名")**:指定したキーが押されたときに**動作(関数)を実行**するように設定。

使用するPythonの標準機能

- **変数への代入と再代入**

例:iy = iy - 10、score = score + 1

- **while True**

終了条件のない無限ループ。

- **time.sleep(0.05)**
  約 0.05 秒(50ミリ秒)ごとに 1 回**更新する間隔を保つ**。入れないと画面更新やキー入力受付が追いつかない。

- **random.randint(a, b)**:

a 以上 b 以下の整数をランダムに返す(例:random.randint(-180, 180))。

- **if と連鎖比較:**
  条件で処理を分岐する。-30 < ix - player.xcor() < 30 は連鎖比較で、(-30 < ix - player.xcor()) and (ix - player.xcor() < 30) と同じ意味。落下物とプレイヤーの中心間の水平距離が 30 未満であることを表す。

使用するPythonの標準機能

- 関数定義

screen.onkey を用いて「キーが押されたとき呼ぶ関数」として設定。

```
def move_left():
    player.setx(player.xcor() - 20)
def move_right():
    player.setx(player.xcor() + 20)

screen.onkey(move_left, "Left")
screen.onkey(move_right, "Right")
```

演習２．落下の再現

- **手順**

https://trinket.io/python/e0c1d1c0f8af

- **動作のようす**

実行すると、画面中央の上端(x=0, y=180)に円が現れ、まっすぐ下へ落ちていきます。下端(y=-180)を過ぎると、また上端に戻って同じ列を落ち続けます。

ヒント

- 落下は、iy を毎回一定量(10)だけ減らして表す(iy = iy - 10)。
- iy が下端(-180)を下回ったら上端(180)に戻すと、落下が繰り返される。
- item.penup() を入れないと、落下のたびに縦線(軌跡)が描かれてしまう。

考察ポイント

- time.sleep(0.05) が更新の間隔(約50ミリ秒)を保っていることを確認する。

Database Lab.

main.py

```
import turtle, time

screen = turtle.Screen()
screen.setup(400, 400)

item = turtle.Turtle("circle")
item.penup()
ix = 0
iy = 180
item.goto(ix, iy)

while True:
    iy = iy - 10
    item.goto(ix, iy)
    if iy < -180:
        iy = 180
    time.sleep(0.05)
```

Result

演習３． 接触の近似

- **手順**

https://trinket.io/python/216f2ddb6b40

- **動作のようす**

下端の中央(x=0, y=-180)に四角いプレイヤーが置かれ、上端から円が落ちてきます。円は毎回 x=0 の同じ列を落ちるため、プレイヤーの真上を通過します。**円が下端に達するたびに、プレイヤーとの水平距離が 30 未満かどうかが判定**され、近ければ score が 1 増えます。プレイヤーはまだ動かせません。

- **ヒント**

- 受け止めの判定は、落下物が下端に達した瞬間(if iy < -180: の中)で行う。
- -30 < ix - player.xcor() < 30 は、中心間の水平距離が 30 未満であることを表す。

```python
# 接触の近似
import turtle, time

screen = turtle.Screen()
screen.setup(400, 400)

player = turtle.Turtle("square")
player.penup()
player.goto(0, -180)

item = turtle.Turtle("circle")
item.penup()
ix = 0
iy = 180
item.goto(ix, iy)

score = 0

while True:
    iy = iy - 10
    item.goto(ix, iy)
    if iy < -180:
        if -30 < ix - player.xcor() < 30:
            score = score + 1
            print(score)
        iy = 180
    time.sleep(0.05)
```

```
3
4
5
6
7
```

演習４．アクションゲーム性

- **手順**

https://trinket.io/python/b2bde1e50f7d

- **動作のようす**

円が落ちてくる左右の位置が毎回ランダムに変わります。最初の出現位置も、下端に達して上端へ戻るときの位置も、-180 から 180 の間でばらばらに決まります。プレイヤーは下端中央に固定されたままなので、**円がたまたまプレイヤーの近く(水平距離 30 未満)に落ちたときだけ得点が入ります**。

ヒント

- random.randint(-180, 180) は -180 以上 180 以下の整数を返す。

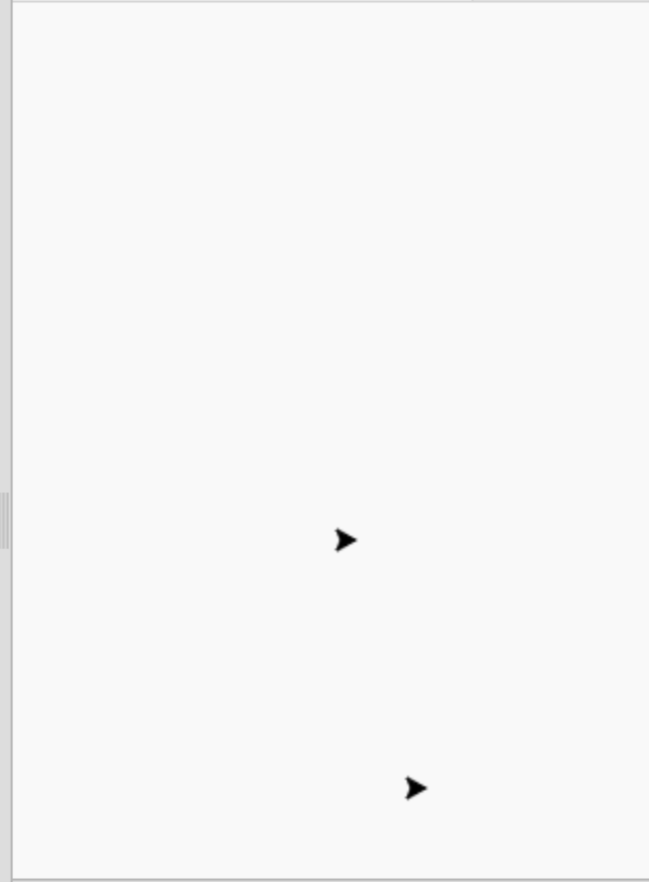
考察ポイント

- 落下量(10)と更新間隔(0.05秒)がゲームのテンポを決めていることを確認する。

main.py

```python
import turtle, random, time

screen = turtle.Screen()
screen.setup(400, 400)

player = turtle.Turtle("square")
player.penup()
player.goto(0, -180)

item = turtle.Turtle("circle")
item.penup()
ix = random.randint(-180, 180)
iy = 180
item.goto(ix, iy)

score = 0

while True:
    iy = iy - 10
    item.goto(ix, iy)
    if iy < -180:
        if -30 < ix - player.xcor() < 30:
            score = score + 1
            print(score)
        ix = random.randint(-180, 180)
        iy = 180
    time.sleep(0.05)
```

Result



1
2
3

演習５．プレイヤーを動かす

- **手順**

https://trinket.io/python/1a7d4df094b4

- **動作のようす**

これまでと同じく円がランダムな位置を落ちてきますが、**今回からプレイヤーを左右キーで動かせます**。左キーを押すとプレイヤーが左へ 20、右キーを押すと右へ 20 だけ動きます。**落ちてくる円に合わせてプレイヤーを移動させ、真下で受け止めると得点になります**。ただし移動範囲の制限がまだ無いので、キーを押し続けると画面の外まで出て行ってしまいます。

遊び方

- **描画領域(タートルの絵が表示される部分)を一度クリックして、フォーカスを与える**(これをしないと矢印キーが効きません)。
- 左矢印キー(←)でプレイヤーが左へ、右矢印キー(→)で右へ動く。
- 落ちてくる円の真下にプレイヤーを移動させ、受け止めて得点をねらう。

考察ポイント

- 左右キーでプレイヤーの x が ±20 変化することを確認する。

```python
import turtle, random, time

screen = turtle.Screen()
screen.setup(400, 400)

player = turtle.Turtle("square")
player.penup()
player.goto(0, -180)

item = turtle.Turtle("circle")
item.penup()
ix = random.randint(-180, 180)
iy = 180
item.goto(ix, iy)

score = 0

def move_left():
    player.setx(player.xcor() - 20)

def move_right():
    player.setx(player.xcor() + 20)

screen.listen()
screen.onkey(move_left, "Left")
screen.onkey(move_right, "Right")

while True:
    iy = iy - 10
    item.goto(ix, iy)
    if iy < -180:
        if -30 < ix - player.xcor() < 30:
            score = score + 1
            print(score)
        ix = random.randint(-180, 180)
        iy = 180
    time.sleep(0.05)
```

```
1
2
```

演習６． 完成形

- **手順**

https://trinket.io/python/826f3b473886

- **動作のようす**

演習5と同じ操作感ですが、プレイヤーが画面の外へ出なくなります。**左へ動かし続けても左端(x=-180)で、右へ動かし続けても右端(x=180)で止まります**。これで、落ちてくる円を追いかけても、プレイヤーが見えなくなってしまうことがなくなります。

遊び方

- 描画領域を一度クリックして、フォーカスを与える。
- 左矢印キー(←)・右矢印キー(→)でプレイヤーを左右に動かす(左右の端で止まる)。
- 落ちてくる円の真下にプレイヤーを移動させ、受け止めて得点をねらう。

ヒント

- 移動関数の中で setx の直後に if で位置を確認し、範囲外なら端の値(±180)に押し戻す。

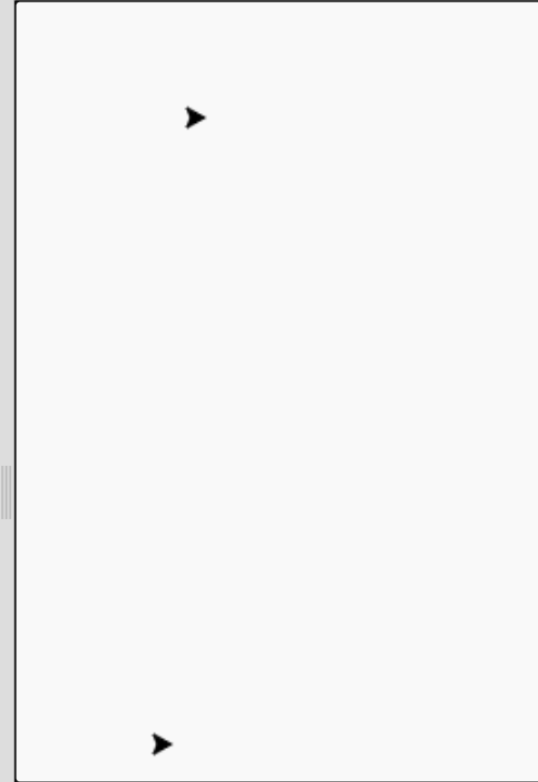
考察ポイント

- 押し戻しによってプレイヤーが左右端(±180)で止まることを確認する。

main.py

```python
player = turtle.Turtle("square")
player.penup()
player.goto(0, -180)

item = turtle.Turtle("circle")
item.penup()
ix = random.randint(-180, 180)
iy = 180
item.goto(ix, iy)

score = 0

def move_left():
    player.setx(player.xcor() - 20)
    if player.xcor() < -180:
        player.setx(-180)

def move_right():
    player.setx(player.xcor() + 20)
    if player.xcor() > 180:
        player.setx(180)

screen.listen()
screen.onkey(move_left, "Left")
screen.onkey(move_right, "Right")

while True:
    iy = iy - 10
    item.goto(ix, iy)
    if iy < -180:
        if -30 < ix - player.xcor() < 30:
            score = score + 1
            print(score)
        ix = random.randint(-180, 180)
        iy = 180
    time.sleep(0.05)
```

Result



```
1
2
```